Modern Java - A Guide to Java 8

wizardforcel

Published with GitBook



Table of Contents

Introduction 0
Modern Java - A Guide to Java 8 1
Java 8 Stream Tutorial 2
Java 8 Nashorn Tutorial 3
Java 8 Concurrency Tutorial: Threads and Executors 4
Java 8 Concurrency Tutorial: Synchronization and Locks 5
Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap 6
Java 8 API by Example: Strings, Numbers, Math and Files 7
Avoiding Null Checks in Java 8 8
Fixing Java 8 Stream Gotchas with IntelliJ IDEA 9
Using Backbone.js with Nashorn 10

Modern Java - A Guide to Java 8

Author: winterbe

From: java8-tutorial

Modern Java - A Guide to Java 8

> "Java is still not dead—and people are starting to figure that out."

Welcome to my introduction to Java 8. This tutorial guides you step by step through all new language features. Backed by short and simple code samples you'll learn how to use default interface methods, lambda expressions, method references and repeatable annotations. At the end of the article you'll be familiar with the most recent API changes like streams, functional interfaces, map extensions and the new Date API. **No walls of text, just a bunch of commented code snippets. Enjoy!**

This article was originally posted on my blog. You should follow me on Twitter.

Table of Contents

- Default Methods for Interfaces
- Lambda expressions
- Functional Interfaces
- Method and Constructor References
- Lambda Scopes
  - Accessing local variables
  - Accessing fields and static variables
  - Accessing Default Interface Methods
- Built-in Functional Interfaces
  - Predicates
  - Functions
  - Suppliers
  - Consumers
  - Comparators
- Optionals
- Streams
  - Filter
  - Sorted
  - Map
  - Match
  - Count
  - Reduce
- Parallel Streams
  - Sequential Sort

- Parallel Sort
- Maps
- Date API
    - Clock
    - Timezones
    - LocalTime
    - LocalDate
    - LocalDateTime
- Annotations
- Where to go from here?

Default Methods for Interfaces

Java 8 enables us to add non-abstract method implementations to interfaces by utilizing the `default` keyword. This feature is also known as virtual extension methods.

Here is our first example:

```
interface Formula {
    double calculate(int a);

    default double sqrt(int a) {
        return Math.sqrt(a);
    }
}
```

Besides the abstract method `calculate` the interface `Formula` also defines the default method `sqrt`. Concrete classes only have to implement the abstract method `calculate`. The default method `sqrt` can be used out of the box.

```
Formula formula = new Formula() {
    @Override
    public double calculate(int a) {
        return sqrt(a * 100);
    }
};

formula.calculate(100);     // 100.0
formula.sqrt(16);           // 4.0
```

The formula is implemented as an anonymous object. The code is quite verbose: 6 lines of code for such a simple calculation of `sqrt(a * 100)`. As we'll see in the next section, there's a much nicer way of implementing single method objects in Java 8.

Lambda expressions

Let's start with a simple example of how to sort a list of strings in prior versions of Java:

```
List<String> names = Arrays.asList("peter", "anna", "mike", "xenia");

Collections.sort(names, new Comparator<String>() {
    @Override
    public int compare(String a, String b) {
        return b.compareTo(a);
    }
});
```

The static utility method `Collections.sort` accepts a list and a comparator in order to sort the elements of the given list. You often find yourself creating anonymous comparators and pass them to the sort method.

Instead of creating anonymous objects all day long, Java 8 comes with a much shorter syntax, **lambda expressions**:

```
Collections.sort(names, (String a, String b) -> {
    return b.compareTo(a);
});
```

As you can see the code is much shorter and easier to read. But it gets even shorter:

```
Collections.sort(names, (String a, String b) -> b.compareTo(a));
```

For one line method bodies you can skip both the braces `{}` and the `return` keyword. But it gets even shorter:

```
names.sort((a, b) -> b.compareTo(a));
```

List now has a `sort` method. Also the java compiler is aware of the parameter types so you can skip them as well. Let's dive deeper into how lambda expressions can be used in the wild.

Functional Interfaces

How does lambda expressions fit into Java's type system? Each lambda corresponds to a given type, specified by an interface. A so called *functional interface* must contain **exactly one abstract method** declaration. Each lambda expression of that type will be matched to this abstract method. Since default methods are not abstract you're free to add default methods to your functional interface.

We can use arbitrary interfaces as lambda expressions as long as the interface only contains one abstract method. To ensure that your interface meet the requirements, you should add the `@FunctionalInterface` annotation. The compiler is aware of this annotation and throws a compiler error as soon as you try to add a second abstract method declaration to the interface.

Example:

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
```

```
Converter<String, Integer> converter = (from) -> Integer.valueOf(from);
Integer converted = converter.convert("123");
System.out.println(converted);    // 123
```

Keep in mind that the code is also valid if the `@FunctionalInterface` annotation would be omitted.

Method and Constructor References

The above example code can be further simplified by utilizing static method references:

```
Converter<String, Integer> converter = Integer::valueOf;
Integer converted = converter.convert("123");
System.out.println(converted);   // 123
```

Java 8 enables you to pass references of methods or constructors via the `::` keyword. The above example shows how to reference a static method. But we can also reference object methods:

```
class Something {
    String startsWith(String s) {
        return String.valueOf(s.charAt(0));
    }
}
```

```
Something something = new Something();
Converter<String, String> converter = something::startsWith;
String converted = converter.convert("Java");
System.out.println(converted);    // "J"
```

Let's see how the `::` keyword works for constructors. First we define an example bean with different constructors:

```
class Person {
    String firstName;
    String lastName;

    Person() {}

    Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Next we specify a person factory interface to be used for creating new persons:

```
interface PersonFactory<P extends Person> {
    P create(String firstName, String lastName);
}
```

Instead of implementing the factory manually, we glue everything together via constructor references:

```
PersonFactory<Person> personFactory = Person::new;
Person person = personFactory.create("Peter", "Parker");
```

We create a reference to the Person constructor via `Person::new`. The Java compiler automatically chooses the right constructor by matching the signature of `PersonFactory.create`.

Lambda Scopes

Accessing outer scope variables from lambda expressions is very similar to anonymous objects. You can access final variables from the local outer scope as well as instance fields and static variables.

Accessing local variables

We can read final local variables from the outer scope of lambda expressions:

```
final int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

But different to anonymous objects the variable `num` does not have to be declared final. This code is also valid:

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);

stringConverter.convert(2);     // 3
```

However `num` must be implicitly final for the code to compile. The following code does **not** compile:

```
int num = 1;
Converter<Integer, String> stringConverter =
        (from) -> String.valueOf(from + num);
num = 3;
```

Writing to `num` from within the lambda expression is also prohibited.

Accessing fields and static variables

In contrast to local variables, we have both read and write access to instance fields and static variables from within lambda expressions. This behaviour is well known from anonymous objects.

```
class Lambda4 {
    static int outerStaticNum;
    int outerNum;

    void testScopes() {
        Converter<Integer, String> stringConverter1 = (from) -> {
            outerNum = 23;
            return String.valueOf(from);
        };

        Converter<Integer, String> stringConverter2 = (from) -> {
            outerStaticNum = 72;
            return String.valueOf(from);
        };
    }
}
```

Accessing Default Interface Methods

Remember the formula example from the first section? Interface `Formula` defines a default method `sqrt` which can be accessed from each formula instance including anonymous objects. This does not work with lambda expressions.

Default methods **cannot** be accessed from within lambda expressions. The following code does not compile:

```
Formula formula = (a) -> sqrt(a * 100);
```

Built-in Functional Interfaces

The JDK 1.8 API contains many built-in functional interfaces. Some of them are well known from older versions of Java like `Comparator` or `Runnable`. Those existing interfaces are extended to enable Lambda support via the `@FunctionalInterface` annotation.

But the Java 8 API is also full of new functional interfaces to make your life easier. Some of those new interfaces are well known from the Google Guava library. Even if you're familiar with this library you should keep a close eye on how those interfaces are extended by some useful method extensions.

Predicates

Predicates are boolean-valued functions of one argument. The interface contains various default methods for composing predicates to complex logical terms (and, or, negate)

```
Predicate<String> predicate = (s) -> s.length() > 0;

predicate.test("foo");              // true
predicate.negate().test("foo");     // false

Predicate<Boolean> nonNull = Objects::nonNull;
Predicate<Boolean> isNull = Objects::isNull;

Predicate<String> isEmpty = String::isEmpty;
Predicate<String> isNotEmpty = isEmpty.negate();
```

Functions

Functions accept one argument and produce a result. Default methods can be used to chain multiple functions together (compose, andThen).

```
Function<String, Integer> toInteger = Integer::valueOf;
Function<String, String> backToString = toInteger.andThen(String::valueOf);

backToString.apply("123");     // "123"
```

Suppliers

Suppliers produce a result of a given generic type. Unlike Functions, Suppliers don't accept arguments.

```
Supplier<Person> personSupplier = Person::new;
personSupplier.get();   // new Person
```

Consumers

Consumers represent operations to be performed on a single input argument.

```
Consumer<Person> greeter = (p) -> System.out.println("Hello, " + p.firstName);
greeter.accept(new Person("Luke", "Skywalker"));
```

Comparators

Comparators are well known from older versions of Java. Java 8 adds various default methods to the interface.

```
Comparator<Person> comparator = (p1, p2) -> p1.firstName.compareTo(p2.firstName);

Person p1 = new Person("John", "Doe");
Person p2 = new Person("Alice", "Wonderland");

comparator.compare(p1, p2);             // > 0
comparator.reversed().compare(p1, p2);  // < 0
```

Optionals

Optionals are not functional interfaces, but nifty utilities to prevent `NullPointerException`. It's an important concept for the next section, so let's have a quick look at how Optionals work.

Optional is a simple container for a value which may be null or non-null. Think of a method which may return a non-null result but sometimes return nothing. Instead of returning `null` you return an `Optional` in Java 8.

```
Optional<String> optional = Optional.of("bam");

optional.isPresent();           // true
optional.get();                 // "bam"
optional.orElse("fallback");    // "bam"

optional.ifPresent((s) -> System.out.println(s.charAt(0)));     // "b"
```

Streams

A `java.util.Stream` represents a sequence of elements on which one or more operations can be performed. Stream operations are either *intermediate* or *terminal*. While terminal operations return a result of a certain type, intermediate operations return the stream itself so you can chain multiple method calls in a row. Streams are created on a source, e.g. a `java.util.Collection` like lists or sets (maps are not supported). Stream operations can either be executed sequentially or parallelly.

> Streams are extremely powerful, so I wrote a separate Java 8 Streams Tutorial. You should also check out Stream.js, a JavaScript port of the Java 8 Streams API.

Let's first look how sequential streams work. First we create a sample source in form of a list of strings:

```
List<String> stringCollection = new ArrayList<>();
stringCollection.add("ddd2");
stringCollection.add("aaa2");
stringCollection.add("bbb1");
stringCollection.add("aaa1");
stringCollection.add("bbb3");
stringCollection.add("ccc");
stringCollection.add("bbb2");
stringCollection.add("ddd1");
```

Collections in Java 8 are extended so you can simply create streams either by calling `Collection.stream()` or `Collection.parallelStream()` . The following sections explain the most common stream operations.

Filter

Filter accepts a predicate to filter all elements of the stream. This operation is *intermediate* which enables us to call another stream operation (`forEach`) on the result. ForEach accepts a consumer to be executed for each element in the filtered stream. ForEach is a terminal operation. It's `void` , so we cannot call another stream operation.

```
stringCollection
    .stream()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa2", "aaa1"
```

Sorted

Sorted is an *intermediate* operation which returns a sorted view of the stream. The elements are sorted in natural order unless you pass a custom `Comparator` .

```
stringCollection
    .stream()
    .sorted()
    .filter((s) -> s.startsWith("a"))
    .forEach(System.out::println);

// "aaa1", "aaa2"
```

Keep in mind that `sorted` does only create a sorted view of the stream without manipulating the ordering of the backed collection. The ordering of `stringCollection` is untouched:

```
System.out.println(stringCollection);
// ddd2, aaa2, bbb1, aaa1, bbb3, ccc, bbb2, ddd1
```

Map

The *intermediate* operation `map` converts each element into another object via the given function. The following example converts each string into an upper-cased string. But you can also use `map` to transform each object into another type. The generic type of the resulting stream depends on the generic type of the function you pass to `map` .

```
stringCollection
    .stream()
    .map(String::toUpperCase)
    .sorted((a, b) -> b.compareTo(a))
    .forEach(System.out::println);

// "DDD2", "DDD1", "CCC", "BBB3", "BBB2", "AAA2", "AAA1"
```

Match

Various matching operations can be used to check whether a certain predicate matches the stream. All of those operations are *terminal* and return a boolean result.

```
boolean anyStartsWithA =
    stringCollection
        .stream()
        .anyMatch((s) -> s.startsWith("a"));

System.out.println(anyStartsWithA);      // true

boolean allStartsWithA =
    stringCollection
        .stream()
        .allMatch((s) -> s.startsWith("a"));

System.out.println(allStartsWithA);      // false

boolean noneStartsWithZ =
    stringCollection
        .stream()
        .noneMatch((s) -> s.startsWith("z"));

System.out.println(noneStartsWithZ);      // true
```

Count

Count is a *terminal* operation returning the number of elements in the stream as a `long` .

```
long startsWithB =
    stringCollection
        .stream()
        .filter((s) -> s.startsWith("b"))
        .count();

System.out.println(startsWithB);    // 3
```

Reduce

This *terminal* operation performs a reduction on the elements of the stream with the given function. The result is an `Optional` holding the reduced value.

```
Optional<String> reduced =
    stringCollection
        .stream()
        .sorted()
        .reduce((s1, s2) -> s1 + "#" + s2);

reduced.ifPresent(System.out::println);
// "aaa1#aaa2#bbb1#bbb2#bbb3#ccc#ddd1#ddd2"
```

Parallel Streams

As mentioned above streams can be either sequential or parallel. Operations on sequential streams are performed on a single thread while operations on parallel streams are performed concurrently on multiple threads.

The following example demonstrates how easy it is to increase the performance by using parallel streams.

First we create a large list of unique elements:

```
int max = 1000000;
List<String> values = new ArrayList<>(max);
for (int i = 0; i < max; i++) {
    UUID uuid = UUID.randomUUID();
    values.add(uuid.toString());
}
```

Now we measure the time it takes to sort a stream of this collection.

Sequential Sort

```
long t0 = System.nanoTime();

long count = values.stream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("sequential sort took: %d ms", millis));

// sequential sort took: 899 ms
```

Parallel Sort

```
long t0 = System.nanoTime();

long count = values.parallelStream().sorted().count();
System.out.println(count);

long t1 = System.nanoTime();

long millis = TimeUnit.NANOSECONDS.toMillis(t1 - t0);
System.out.println(String.format("parallel sort took: %d ms", millis));

// parallel sort took: 472 ms
```

As you can see both code snippets are almost identical but the parallel sort is roughly 50% faster. All you have to do is change `stream()` to `parallelStream()` .

Maps

As already mentioned maps do not directly support streams. There's no `stream()` method available on the `Map` interface itself, however you can create specialized streams upon the keys, values or entries of a map via `map.keySet().stream()` , `map.values().stream()` and `map.entrySet().stream()` .

Furthermore maps support various new and useful methods for doing common tasks.

```
Map<Integer, String> map = new HashMap<>();

for (int i = 0; i < 10; i++) {
    map.putIfAbsent(i, "val" + i);
}

map.forEach((id, val) -> System.out.println(val));
```

The above code should be self-explaining: `putIfAbsent` prevents us from writing additional if null checks; `forEach` accepts a consumer to perform operations for each value of the map.

This example shows how to compute code on the map by utilizing functions:

```
map.computeIfPresent(3, (num, val) -> val + num);
map.get(3);             // val33

map.computeIfPresent(9, (num, val) -> null);
map.containsKey(9);     // false

map.computeIfAbsent(23, num -> "val" + num);
map.containsKey(23);    // true

map.computeIfAbsent(3, num -> "bam");
map.get(3);             // val33
```

Next, we learn how to remove entries for a given key, only if it's currently mapped to a given value:

```
map.remove(3, "val3");
map.get(3);             // val33

map.remove(3, "val33");
map.get(3);             // null
```

Another helpful method:

```
map.getOrDefault(42, "not found");  // not found
```

Merging entries of a map is quite easy:

```
map.merge(9, "val9", (value, newValue) -> value.concat(newValue));
map.get(9);             // val9

map.merge(9, "concat", (value, newValue) -> value.concat(newValue));
map.get(9);             // val9concat
```

Merge either put the key/value into the map if no entry for the key exists, or the merging function will be called to change the existing value.

Date API

Java 8 contains a brand new date and time API under the package `java.time`. The new Date API is comparable with the Joda-Time library, however it's not the same. The following examples cover the most important parts of this new API.

Clock

Clock provides access to the current date and time. Clocks are aware of a timezone and may be used instead of `System.currentTimeMillis()` to retrieve the current time in milliseconds since Unix EPOCH. Such an instantaneous point on the time-line is also represented by the class `Instant`. Instants can be used to create legacy `java.util.Date` objects.

```
Clock clock = Clock.systemDefaultZone();
long millis = clock.millis();

Instant instant = clock.instant();
Date legacyDate = Date.from(instant);   // legacy java.util.Date
```

Timezones

Timezones are represented by a `ZoneId`. They can easily be accessed via static factory methods. Timezones define the offsets which are important to convert between instants and local dates and times.

```
System.out.println(ZoneId.getAvailableZoneIds());
// prints all available timezone ids

ZoneId zone1 = ZoneId.of("Europe/Berlin");
ZoneId zone2 = ZoneId.of("Brazil/East");
System.out.println(zone1.getRules());
System.out.println(zone2.getRules());

// ZoneRules[currentStandardOffset=+01:00]
// ZoneRules[currentStandardOffset=-03:00]
```

LocalTime

LocalTime represents a time without a timezone, e.g. 10pm or 17:30:15. The following example creates two local times for the timezones defined above. Then we compare both times and calculate the difference in hours and minutes between both times.

```
LocalTime now1 = LocalTime.now(zone1);
LocalTime now2 = LocalTime.now(zone2);

System.out.println(now1.isBefore(now2));  // false

long hoursBetween = ChronoUnit.HOURS.between(now1, now2);
long minutesBetween = ChronoUnit.MINUTES.between(now1, now2);

System.out.println(hoursBetween);       // -3
System.out.println(minutesBetween);     // -239
```

LocalTime comes with various factory methods to simplify the creation of new instances, including parsing of time strings.

```
LocalTime late = LocalTime.of(23, 59, 59);
System.out.println(late);       // 23:59:59

DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedTime(FormatStyle.SHORT)
        .withLocale(Locale.GERMAN);

LocalTime leetTime = LocalTime.parse("13:37", germanFormatter);
System.out.println(leetTime);   // 13:37
```

LocalDate

LocalDate represents a distinct date, e.g. 2014-03-11. It's immutable and works exactly analog to LocalTime. The sample demonstrates how to calculate new dates by adding or subtracting days, months or years. Keep in mind that each manipulation returns a new instance.

```
LocalDate today = LocalDate.now();
LocalDate tomorrow = today.plus(1, ChronoUnit.DAYS);
LocalDate yesterday = tomorrow.minusDays(2);

LocalDate independenceDay = LocalDate.of(2014, Month.JULY, 4);
DayOfWeek dayOfWeek = independenceDay.getDayOfWeek();
System.out.println(dayOfWeek);    // FRIDAY
```

Parsing a LocalDate from a string is just as simple as parsing a LocalTime:

```
DateTimeFormatter germanFormatter =
    DateTimeFormatter
        .ofLocalizedDate(FormatStyle.MEDIUM)
        .withLocale(Locale.GERMAN);

LocalDate xmas = LocalDate.parse("24.12.2014", germanFormatter);
System.out.println(xmas);   // 2014-12-24
```

LocalDateTime

LocalDateTime represents a date-time. It combines date and time as seen in the above sections into one instance. `LocalDateTime` is immutable and works similar to LocalTime and LocalDate. We can utilize methods for retrieving certain fields from a date-time:

```
LocalDateTime sylvester = LocalDateTime.of(2014, Month.DECEMBER, 31, 23, 59, 59);

DayOfWeek dayOfWeek = sylvester.getDayOfWeek();
System.out.println(dayOfWeek);      // WEDNESDAY

Month month = sylvester.getMonth();
System.out.println(month);          // DECEMBER

long minuteOfDay = sylvester.getLong(ChronoField.MINUTE_OF_DAY);
System.out.println(minuteOfDay);    // 1439
```

With the additional information of a timezone it can be converted to an instant. Instants can easily be converted to legacy dates of type `java.util.Date`.

```
Instant instant = sylvester
        .atZone(ZoneId.systemDefault())
        .toInstant();

Date legacyDate = Date.from(instant);
System.out.println(legacyDate);     // Wed Dec 31 23:59:59 CET 2014
```

Formatting date-times works just like formatting dates or times. Instead of using pre-defined formats we can create formatters from custom patterns.

```
DateTimeFormatter formatter =
    DateTimeFormatter
        .ofPattern("MMM dd, yyyy - HH:mm");

LocalDateTime parsed = LocalDateTime.parse("Nov 03, 2014 - 07:13", formatter);
String string = formatter.format(parsed);
System.out.println(string);     // Nov 03, 2014 - 07:13
```

Unlike `java.text.NumberFormat` the new `DateTimeFormatter` is immutable and **thread-safe**.

For details on the pattern syntax read here.

Annotations

Annotations in Java 8 are repeatable. Let's dive directly into an example to figure that out.

First, we define a wrapper annotation which holds an array of the actual annotations:

```
@interface Hints {
    Hint[] value();
}

@Repeatable(Hints.class)
@interface Hint {
    String value();
}
```

Java 8 enables us to use multiple annotations of the same type by declaring the annotation `@Repeatable`.

Variant 1: Using the container annotation (old school)

```
@Hints({@Hint("hint1"), @Hint("hint2")})
class Person {}
```

Variant 2: Using repeatable annotations (new school)

```
@Hint("hint1")
@Hint("hint2")
class Person {}
```

Using variant 2 the java compiler implicitly sets up the `@Hints` annotation under the hood. That's important for reading annotation information via reflection.

```
Hint hint = Person.class.getAnnotation(Hint.class);
System.out.println(hint);                   // null

Hints hints1 = Person.class.getAnnotation(Hints.class);
System.out.println(hints1.value().length);  // 2

Hint[] hints2 = Person.class.getAnnotationsByType(Hint.class);
System.out.println(hints2.length);          // 2
```

Although we never declared the `@Hints` annotation on the `Person` class, it's still readable via `getAnnotation(Hints.class)`. However, the more convenient method is `getAnnotationsByType` which grants direct access to all annotated `@Hint` annotations.

Furthermore the usage of annotations in Java 8 is expanded to two new targets:

```
@Target({ElementType.TYPE_PARAMETER, ElementType.TYPE_USE})
@interface MyAnnotation {}
```

Where to go from here?

My programming guide to Java 8 ends here. If you want to learn more about all the new classes and features of the JDK 8 API, check out my JDK8 API Explorer. It helps you figuring out all the new classes and hidden gems of JDK 8, like `Arrays.parallelSort`, `StampedLock` and `CompletableFuture` - just to name a few.

I've also published a bunch of follow-up articles on my blog that might be interesting to you:

- Java 8 Stream Tutorial
- Java 8 Nashorn Tutorial
- Java 8 Concurrency Tutorial: Threads and Executors
- Java 8 Concurrency Tutorial: Synchronization and Locks
- Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap
- Java 8 API by Example: Strings, Numbers, Math and Files
- Avoid Null Checks in Java 8
- Fixing Java 8 Stream Gotchas with IntelliJ IDEA
- Using Backbone.js with Java 8 Nashorn

You should follow me on Twitter. Thanks for reading!

Java 8 Stream Tutorial

July 31, 2014

This example-driven tutorial gives an in-depth overview about Java 8 streams. When I first read about the `Stream` API, I was confused about the name since it sounds similar to `InputStream` and `OutputStream` from Java I/O. But Java 8 streams are a completely different thing. Streams are Monads, thus playing a big part in bringing *functional programming* to Java:

> In functional programming, a monad is a structure that represents computations defined as sequences of steps. A type with a monad structure defines what it means to chain operations, or nest functions of that type together.

This guide teaches you how to work with Java 8 streams and how to use the different kind of available stream operations. You'll learn about the processing order and how the ordering of stream operations affect runtime performance. The more powerful stream operations `reduce`, `collect` and `flatMap` are covered in detail. The tutorial ends with an in-depth look at parallel streams.

If you're not yet familiar with Java 8 lambda expressions, functional interfaces and method references, you probably want to read my Java 8 Tutorial first before starting with this tutorial.

UPDATE *- I'm currently working on a JavaScript implementation of the Java 8 Streams API for the browser. If I've drawn your interest check out Stream.js on GitHub. Your Feedback is highly appreciated.*

How streams work

A stream represents a sequence of elements and supports different kind of operations to perform computations upon those elements:

```
List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

// C1
// C2
```

Stream operations are either intermediate or terminal. Intermediate operations return a stream so we can chain multiple intermediate operations without using semicolons. Terminal operations are either void or return a non-stream result. In the above example `filter`, `map` and `sorted` are intermediate operations whereas `forEach` is a terminal operation. For a full list of all available stream operations see the Stream Javadoc. Such a chain of stream operations as seen in the example above is also known as *operation pipeline*.

Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both *non-interfering* and *stateless*. What does that mean?

A function is non-interfering when it does not modify the underlying data source of the stream, e.g. in the above example no lambda expression does modify `myList` by adding or removing elements from the collection.

A function is stateless when the execution of the operation is deterministic, e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

Different kind of streams

Streams can be created from various data sources, especially collections. Lists and Sets support new methods `stream()` and `parallelStream()` to either create a sequential or a parallel stream. Parallel streams are capable of operating on multiple threads and will be covered in a later section of this tutorial. We focus on sequential streams for now:

```
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

Calling the method `stream()` on a list of objects returns a regular object stream. But we don't have to create collections in order to work with streams as we see in the next code sample:

```
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);  // a1
```

Just use `Stream.of()` to create a stream from a bunch of object references.

Besides regular object streams Java 8 ships with special kinds of streams for working with the primitive data types `int`, `long` and `double`. As you might have guessed it's `IntStream`, `LongStream` and `DoubleStream`.

IntStreams can replace the regular for-loop utilizing `IntStream.range()` :

```
IntStream.range(1, 4)
    .forEach(System.out::println);

// 1
// 2
// 3
```

All those primitive streams work just like regular object streams with the following differences: Primitive streams use specialized lambda expressions, e.g. `IntFunction` instead of `Function` or `IntPredicate` instead of `Predicate` . And primitive streams support the additional terminal aggregate operations `sum()` and `average()` :

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println);  // 5.0
```

Sometimes it's useful to transform a regular object stream to a primitive stream or vice versa. For that purpose object streams support the special mapping operations `mapToInt()` , `mapToLong()` and `mapToDouble` :

```
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

Primitive streams can be transformed to object streams via `mapToObj()` :

```
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

Here's a combined example: the stream of doubles is first mapped to an int stream and than mapped to an object stream of strings:

```
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

Processing Order

Now that we've learned how to create and work with different kinds of streams, let's dive deeper into how stream operations are processed under the hood.

An important characteristic of intermediate operations is laziness. Look at this sample where a terminal operation is missing:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    });
```

When executing this code snippet, nothing is printed to the console. That is because intermediate operations will only be executed when a terminal operation is present.

Let's extend the above example by the terminal operation `forEach` :

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return true;
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Executing this code snippet results in the desired output on the console:

```
filter:  d2
forEach: d2
filter:  a2
forEach: a2
filter:  b1
forEach: b1
filter:  b3
forEach: b3
filter:  c
forEach: c
```

The order of the result might be surprising. A naive approach would be to execute the operations horizontally one after another on all elements of the stream. But instead each element moves along the chain vertically. The first string "d2" passes `filter` then `forEach` , only then the second string "a2" is processed.

This behavior can reduce the actual number of operations performed on each element, as we see in the next example:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .anyMatch(s -> {
        System.out.println("anyMatch: " + s);
        return s.startsWith("A");
    });

// map:      d2
// anyMatch: D2
// map:      a2
// anyMatch: A2
```

The operation `anyMatch` returns `true` as soon as the predicate applies to the given input element. This is true for the second element passed "A2". Due to the vertical execution of the stream chain, `map` has only to be executed twice in this case. So instead of mapping all elements of the stream, `map` will be called as few as possible.

Why order matters

The next example consists of two intermediate operations `map` and `filter` and the terminal operation `forEach`. Let's once again inspect how those operations are being executed:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
    .forEach(s -> System.out.println("forEach: " + s));

// map:     d2
// filter:  D2
// map:     a2
// filter:  A2
// forEach: A2
// map:     b1
// filter:  B1
// map:     b3
// filter:  B3
// map:     c
// filter:  C
```

As you might have guessed both `map` and `filter` are called five times for every string in the underlying collection whereas `forEach` is only called once.

We can greatly reduce the actual number of executions if we change the order of the operations, moving `filter` to the beginning of the chain:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// map:     a2
// forEach: A2
// filter:  b1
// filter:  b3
// filter:  c
```

Now, `map` is only called once so the operation pipeline performs much faster for larger numbers of input elements. Keep that in mind when composing complex method chains.

Let's extend the above example by an additional operation, `sorted` :

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
```

Sorting is a special kind of intermediate operation. It's a so called *stateful operation* since in order to sort a collection of elements you have to maintain state during ordering.

Executing this example results in the following console output:

```
sort:    a2; d2
sort:    b1; a2
sort:    b1; d2
sort:    b1; a2
sort:    b3; b1
sort:    b3; d2
sort:    c; b3
sort:    c; d2
filter:  a2
map:     a2
forEach: A2
filter:  b1
filter:  b3
filter:  c
filter:  d2
```

First, the sort operation is executed on the entire input collection. In other words `sorted` is executed horizontally. So in this case `sorted` is called eight times for multiple combinations on every element in the input collection.

Once again we can optimize the performance by reordering the chain:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .sorted((s1, s2) -> {
        System.out.printf("sort: %s; %s\n", s1, s2);
        return s1.compareTo(s2);
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));

// filter:  d2
// filter:  a2
// filter:  b1
// filter:  b3
// filter:  c
// map:     a2
// forEach: A2
```

In this example `sorted` is never been called because `filter` reduces the input collection to just one element. So the performance is greatly increased for larger input collections.

Reusing Streams

Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed:

```
Stream<String> stream =
    Stream.of("d2", "a2", "b1", "b3", "c")
        .filter(s -> s.startsWith("a"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

Calling `noneMatch` after `anyMatch` on the same stream results in the following exception:

```
java.lang.IllegalStateException: stream has already been operated upon or closed
    at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
    at java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
    at com.winterbe.java8.Streams5.test7(Streams5.java:38)
    at com.winterbe.java8.Streams5.main(Streams5.java:28)
```

To overcome this limitation we have to to create a new stream chain for every terminal operation we want to execute, e.g. we could create a stream supplier to construct a new stream with all intermediate operations already set up:

```
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
            .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

Each call to `get()` constructs a new stream on which we are save to call the desired terminal operation.

Advanced Operations

Streams support plenty of different operations. We've already learned about the most important operations like `filter` or `map`. I leave it up to you to discover all other available operations (see Stream Javadoc). Instead let's dive deeper into the more complex operations `collect`, `flatMap` and `reduce`.

Most code samples from this section use the following list of persons for demonstration purposes:

```
class Person {
    String name;
    int age;

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return name;
    }
}

List<Person> persons =
    Arrays.asList(
        new Person("Max", 18),
        new Person("Peter", 23),
        new Person("Pamela", 23),
        new Person("David", 12));
```

Collect

Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a `List`, `Set` or `Map`. Collect accepts a `Collector` which consists of four different operations: a *supplier*, an *accumulator*, a *combiner* and a *finisher*.

This sounds super complicated at first, but the good part is Java 8 supports various built-in collectors via the `Collectors` class. So for the most common operations you don't have to implement a collector yourself.

Let's start with a very common usecase:

```
List<Person> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("P"))
        .collect(Collectors.toList());

System.out.println(filtered);    // [Peter, Pamela]
```

As you can see it's very simple to construct a list from the elements of a stream. Need a set instead of list - just use `Collectors.toSet()`.

The next example groups all persons by age:

```
Map<Integer, List<Person>> personsByAge = persons
    .stream()
    .collect(Collectors.groupingBy(p -> p.age));

personsByAge
    .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));

// age 18: [Max]
// age 23: [Peter, Pamela]
// age 12: [David]
```

Collectors are extremely versatile. You can also create aggregations on the elements of the stream, e.g. determining the average age of all persons:

```
Double averageAge = persons
    .stream()
    .collect(Collectors.averagingInt(p -> p.age));

System.out.println(averageAge);     // 19.0
```

If you're interested in more comprehensive statistics, the summarizing collectors return a special built-in summary statistics object. So we can simply determine *min*, *max* and arithmetic *average* age of the persons as well as the *sum* and *count*.

```
IntSummaryStatistics ageSummary =
    persons
        .stream()
        .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

The next example joins all persons into a single string:

```
String phrase = persons
    .stream()
    .filter(p -> p.age >= 18)
    .map(p -> p.name)
    .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

The join collector accepts a delimiter as well as an optional prefix and suffix.

In order to transform the stream elements into a map, we have to specify how both the keys and the values should be mapped. Keep in mind that the mapped keys must be unique, otherwise an `IllegalStateException` is thrown. You can optionally pass a merge function as an additional parameter to bypass the exception:

```
Map<Integer, String> map = persons
    .stream()
    .collect(Collectors.toMap(
        p -> p.age,
        p -> p.name,
        (name1, name2) -> name1 + ";" + name2));

System.out.println(map);
// {18=Max, 23=Peter;Pamela, 12=David}
```

Now that we know some of the most powerful built-in collectors, let's try to build our own special collector. We want to transform all persons of the stream into a single string consisting of all names in upper letters separated by the `|` pipe character. In order to achieve this we create a new collector via `Collector.of()`. We have to pass the four ingredients of a collector: a *supplier*, an *accumulator*, a *combiner* and a *finisher*.

```
Collector<Person, StringJoiner, String> personNameCollector =
    Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
    .stream()
    .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID
```

Since strings in Java are immutable, we need a helper class like `StringJoiner` to let the collector construct our string. The supplier initially constructs such a StringJoiner with the appropriate delimiter. The accumulator is used to add each persons upper-cased name to the StringJoiner. The combiner knows how to merge two StringJoiners into one. In the last step the finisher constructs the desired String from the StringJoiner.

FlatMap

We've already learned how to transform the objects of a stream into another type of objects by utilizing the `map` operation. Map is kinda limited because every object can only be mapped to exactly one other object. But what if we want to transform one object into multiple others or none at all? This is where `flatMap` comes to the rescue.

FlatMap transforms each element of the stream into a stream of other objects. So each object will be transformed into zero, one or multiple other objects backed by streams. The contents of those streams will then be placed into the returned stream of the `flatMap` operation.

Before we see `flatMap` in action we need an appropriate type hierarchy:

```
class Foo {
    String name;
    List<Bar> bars = new ArrayList<>();

    Foo(String name) {
        this.name = name;
    }
}

class Bar {
    String name;

    Bar(String name) {
        this.name = name;
    }
}
```

Next, we utilize our knowledge about streams to instantiate a couple of objects:

```
List<Foo> foos = new ArrayList<>();

// create foos
IntStream
    .range(1, 4)
    .forEach(i -> foos.add(new Foo("Foo" + i)));

// create bars
foos.forEach(f ->
    IntStream
        .range(1, 4)
        .forEach(i -> f.bars.add(new Bar("Bar" + i + " <- " + f.name))));
```

Now we have a list of three foos each consisting of three bars.

FlatMap accepts a function which has to return a stream of objects. So in order to resolve the bar objects of each foo, we just pass the appropriate function:

```
foos.stream()
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));

// Bar1 <- Foo1
// Bar2 <- Foo1
// Bar3 <- Foo1
// Bar1 <- Foo2
// Bar2 <- Foo2
// Bar3 <- Foo2
// Bar1 <- Foo3
// Bar2 <- Foo3
// Bar3 <- Foo3
```

As you can see, we've successfully transformed the stream of three foo objects into a stream of nine bar objects.

Finally, the above code example can be simplified into a single pipeline of stream operations:

```
IntStream.range(1, 4)
    .mapToObj(i -> new Foo("Foo" + i))
    .peek(f -> IntStream.range(1, 4)
        .mapToObj(i -> new Bar("Bar" + i + " <- " f.name))
        .forEach(f.bars::add))
    .flatMap(f -> f.bars.stream())
    .forEach(b -> System.out.println(b.name));
```

FlatMap is also available for the `Optional` class introduced in Java 8. Optionals `flatMap` operation returns an optional object of another type. So it can be utilized to prevent nasty `null` checks.

Think of a highly hierarchical structure like this:

```
class Outer {
    Nested nested;
}

class Nested {
    Inner inner;
}

class Inner {
    String foo;
}
```

In order to resolve the inner string `foo` of an outer instance you have to add multiple null checks to prevent possible NullPointerExceptions:

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner != null) {
    System.out.println(outer.nested.inner.foo);
}
```

The same behavior can be obtained by utilizing optionals `flatMap` operation:

```
Optional.of(new Outer())
    .flatMap(o -> Optional.ofNullable(o.nested))
    .flatMap(n -> Optional.ofNullable(n.inner))
    .flatMap(i -> Optional.ofNullable(i.foo))
    .ifPresent(System.out::println);
```

Each call to `flatMap` returns an `Optional` wrapping the desired object if present or `null` if absent.

Reduce

The reduction operation combines all elements of the stream into a single result. Java 8 supports three different kind of `reduce` methods. The first one reduces a stream of elements to exactly one element of the stream. Let's see how we can use this method to determine the oldest person:

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);    // Pamela
```

The `reduce` method accepts a `BinaryOperator` accumulator function. That's actually a `BiFunction` where both operands share the same type, in that case `Person`. BiFunctions are like `Function` but accept two arguments. The example function compares both persons ages in order to return the person with the maximum age.

The second `reduce` method accepts both an identity value and a `BinaryOperator` accumulator. This method can be utilized to construct a new Person with the aggregated names and ages from all other persons in the stream:

```
Person result =
    persons
        .stream()
        .reduce(new Person("", 0), (p1, p2) -> {
            p1.age += p2.age;
            p1.name += p2.name;
            return p1;
        });

System.out.format("name=%s; age=%s", result.name, result.age);
// name=MaxPeterPamelaDavid; age=76
```

The third `reduce` method accepts three parameters: an identity value, a `BiFunction` accumulator and a combiner function of type `BinaryOperator`. Since the identity values type is not restricted to the `Person` type, we can utilize this reduction to determine the sum of ages from all persons:

```
Integer ageSum = persons
    .stream()
    .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1 + sum2);

System.out.println(ageSum);  // 76
```

As you can see the result is *76*, but what's happening exactly under the hood? Let's extend the above code by some debug output:

```
Integer ageSum = persons
    .stream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
            return sum1 + sum2;
        });

// accumulator: sum=0; person=Max
// accumulator: sum=18; person=Peter
// accumulator: sum=41; person=Pamela
// accumulator: sum=64; person=David
```

As you can see the accumulator function does all the work. It first get called with the initial identity value *0* and the first person *Max*. In the next three steps `sum` continually increases by the age of the last steps person up to a total age of *76*.

Wait wat? The combiner never gets called? Executing the same stream in parallel will lift the secret:

```
Integer ageSum = persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s\n", sum, p);
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s\n", sum1, sum2);
            return sum1 + sum2;
        });

// accumulator: sum=0; person=Pamela
// accumulator: sum=0; person=David
// accumulator: sum=0; person=Max
// accumulator: sum=0; person=Peter
// combiner: sum1=18; sum2=23
// combiner: sum1=23; sum2=12
// combiner: sum1=41; sum2=35
```

Executing this stream in parallel results in an entirely different execution behavior. Now the combiner is actually called. Since the accumulator is called in parallel, the combiner is needed to sum up the separate accumulated values.

Let's dive deeper into parallel streams in the next chapter.

Parallel Streams

Streams can be executed in parallel to increase runtime performance on large amount of input elements. Parallel streams use a common `ForkJoinPool` available via the static `ForkJoinPool.commonPool()` method. The size of the underlying thread-pool uses up to five threads - depending on the amount of available physical CPU cores:

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism());    // 3
```

On my machine the common pool is initialized with a parallelism of 3 per default. This value can be decreased or increased by setting the following JVM parameter:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

Collections support the method `parallelStream()` to create a parallel stream of elements. Alternatively you can call the intermediate method `parallel()` on a given stream to convert a sequential stream to a parallel counterpart.

In order to understate the parallel execution behavior of a parallel stream the next example prints information about the current thread to `sout`:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

By investigating the debug output we should get a better understanding which threads are actually used to execute the stream operations:

```
filter:  b1 [main]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  c2 [ForkJoinPool.commonPool-worker-3]
map:     c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map:     b1 [main]
forEach: B1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-3]
map:     a1 [ForkJoinPool.commonPool-worker-3]
forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

As you can see the parallel stream utilizes all available threads from the common `ForkJoinPool` for executing the stream operations. The output may differ in consecutive runs because the behavior which particular thread is actually used is non-deterministic.

Let's extend the example by an additional stream operation, `sort` :

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
        System.out.format("sort: %s <> %s [%s]\n",
            s1, s2, Thread.currentThread().getName());
        return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

The result may look strange at first:

```
filter:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:     c1 [ForkJoinPool.commonPool-worker-2]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:     a2 [ForkJoinPool.commonPool-worker-1]
filter:  b1 [main]
map:     b1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-2]
map:     a1 [ForkJoinPool.commonPool-worker-2]
map:     c2 [ForkJoinPool.commonPool-worker-3]
sort:    A2 <> A1 [main]
sort:    B1 <> A2 [main]
sort:    C2 <> B1 [main]
sort:    C1 <> C2 [main]
sort:    C1 <> B1 [main]
sort:    C1 <> C2 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-1]
```

It seems that `sort` is executed sequentially on the main thread only. Actually, `sort` on a parallel stream uses the new Java 8 method `Arrays.parallelSort()` under the hood. As stated in Javadoc this method decides on the length of the array if sorting will be performed sequentially or in parallel:

> If the length of the specified array is less than the minimum granularity, then it is sorted using the appropriate Arrays.sort method.

Coming back to the `reduce` example from the last section. We already found out that the combiner function is only called in parallel but not in sequential streams. Let's see which threads are actually involved:

```
List<Person> persons = Arrays.asList(
    new Person("Max", 18),
    new Person("Peter", 23),
    new Person("Pamela", 23),
    new Person("David", 12));

persons
    .parallelStream()
    .reduce(0,
        (sum, p) -> {
            System.out.format("accumulator: sum=%s; person=%s [%s]\n",
                sum, p, Thread.currentThread().getName());
            return sum += p.age;
        },
        (sum1, sum2) -> {
            System.out.format("combiner: sum1=%s; sum2=%s [%s]\n",
                sum1, sum2, Thread.currentThread().getName());
            return sum1 + sum2;
        });
```

The console output reveals that both the *accumulator* and the *combiner* functions are executed in parallel on all available threads:

```
accumulator: sum=0; person=Pamela; [main]
accumulator: sum=0; person=Max;    [ForkJoinPool.commonPool-worker-3]
accumulator: sum=0; person=David;  [ForkJoinPool.commonPool-worker-2]
accumulator: sum=0; person=Peter;  [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=18; sum2=23;     [ForkJoinPool.commonPool-worker-1]
combiner:    sum1=23; sum2=12;     [ForkJoinPool.commonPool-worker-2]
combiner:    sum1=41; sum2=35;     [ForkJoinPool.commonPool-worker-2]
```

In summary, it can be stated that parallel streams can bring be a nice performance boost to streams with a large amount of input elements. But keep in mind that some parallel stream operations like `reduce` and `collect` need additional computations (combine operations) which isn't needed when executed sequentially.

Furthermore we've learned that all parallel stream operations share the same JVM-wide common `ForkJoinPool`. So you probably want to avoid implementing slow blocking stream operations since that could potentially slow down other parts of your application which rely heavily on parallel streams.

That's it

My programming guide to Java 8 streams ends here. If you're interested in learning more about Java 8 streams, I recommend to you the Stream Javadoc package documentation. If you want to learn more about the underlying mechanisms, you probably want to read Martin Fowlers article about Collection Pipelines.

If you're interested in JavaScript as well, you may want to have a look at Stream.js - a JavaScript implementation of the Java 8 Streams API. You may also wanna read my Java 8 Tutorial and my Java 8 Nashorn Tutorial.

Hopefully this tutorial was helpful to you and you've enjoyed reading it. The full source code of the tutorial samples is hosted on GitHub. Feel free to fork the repository or send me your feedback via Twitter.

Happy coding!

Java 8 Nashorn Tutorial

April 05, 2014

Learn all about the Nashorn Javascript Engine with easily understood code examples. The Nashorn Javascript Engine is part of Java SE 8 and competes with other standalone engines like Google V8 (the engine that powers Google Chrome and Node.js). Nashorn extends Javas capabilities by running dynamic javascript code natively on the JVM.

In the next ~15 minutes you learn how to evaluate javascript on the JVM dynamically during runtime. The most recent Nashorn language features are demonstrated with small code examples. You learn how to call javascript functions from java code and vice versa. At the end you're ready to integrate dynamic scripts in your daily java business.



UPDATE *- I'm currently working on a JavaScript implementation of the Java 8 Streams API for the browser. If I've drawn your interest check out Stream.js on GitHub. Your Feedback is highly appreciated.*

Using Nashorn

The Nashorn javascript engine can either be used programmatically from java programs or by utilizing the command line tool `jjs`, which is located in `$JAVA_HOME/bin`. If you plan to work with `jjs` you might want to put a symbolic link for simple access:

```
$ cd /usr/bin
$ ln -s $JAVA_HOME/bin/jjs jjs
$ jjs
jjs> print('Hello World');
```

This tutorial focuses on using nashorn from java code, so let's skip `jjs` for now. A simple HelloWorld in java code looks like this:

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval("print('Hello World!');");
```

In order to evaluate javascript code from java, you first create a nashorn script engine by utilizing the `javax.script` package already known from Rhino (Javas legacy js engine from Mozilla).

Javascript code can either be evaluated directly by passing javascript code as a string as shown above. Or you can pass a file reader pointing to your .js script file:

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("script.js"));
```

Nashorn javascript is based on ECMAScript 5.1 but future versions of nashorn will include support for ECMAScript 6:

> The current strategy for Nashorn is to follow the ECMAScript specification. When we release with JDK 8 we will be aligned with ECMAScript 5.1. The follow up major release of Nashorn will align with ECMAScript Edition 6.

Nashorn defines a lot of language and API extensions to the ECMAScript standard. But first let's take a look at how the communication between java and javascript code works.

Invoking Javascript Functions from Java

Nashorn supports the invocation of javascript functions defined in your script files directly from java code. You can pass java objects as function arguments and return data back from the function to the calling java method.

The following javascript functions will later be called from the java side:

```
var fun1 = function(name) {
    print('Hi there from Javascript, ' + name);
    return "greetings from javascript";
};

var fun2 = function (object) {
    print("JS Class Definition: " + Object.prototype.toString.call(object));
};
```

In order to call a function you first have to cast the script engine to `Invocable`. The Invocable interface is implemented by the `NashornScriptEngine` implementation and defines a method `invokeFunction` to call a javascript function for a given name.

```
ScriptEngine engine = new ScriptEngineManager().getEngineByName("nashorn");
engine.eval(new FileReader("script.js"));

Invocable invocable = (Invocable) engine;

Object result = invocable.invokeFunction("fun1", "Peter Parker");
System.out.println(result);
System.out.println(result.getClass());

// Hi there from Javascript, Peter Parker
// greetings from javascript
// class java.lang.String
```

Executing the code results in three lines written to the console. Calling the function `print` pipes the result to `System.out`, so we see the javascript message first.

Now let's call the second function by passing arbitrary java objects:

```
invocable.invokeFunction("fun2", new Date());
// [object java.util.Date]

invocable.invokeFunction("fun2", LocalDateTime.now());
// [object java.time.LocalDateTime]

invocable.invokeFunction("fun2", new Person());
// [object com.winterbe.java8.Person]
```

Java objects can be passed without loosing any type information on the javascript side. Since the script runs natively on the JVM we can utilize the full power of the Java API or external libraries on nashorn.

Invoking Java Methods from Javascript

Invoking java methods from javascript is quite easy. We first define a static java method:

```
static String fun1(String name) {
    System.out.format("Hi there from Java, %s", name);
    return "greetings from java";
}
```

Java classes can be referenced from javascript via the `Java.type` API extension. It's similar to importing classes in java code. As soon as the java type is defined we naturally call the static method `fun1()` and print the result to `sout`. Since the method is static, we don't have to create an instance first.

```
var MyJavaClass = Java.type('my.package.MyJavaClass');

var result = MyJavaClass.fun1('John Doe');
print(result);

// Hi there from Java, John Doe
// greetings from java
```

How does Nashorn handle type conversion when calling java methods with native javascript types? Let's find out with a simple example.

The following java method simply prints the actual class type of the method parameter:

```
static void fun2(Object object) {
    System.out.println(object.getClass());
}
```

To understand how type conversations are handled under the hood, we call this method with different javascript types:

```
MyJavaClass.fun2(123);
// class java.lang.Integer

MyJavaClass.fun2(49.99);
// class java.lang.Double

MyJavaClass.fun2(true);
// class java.lang.Boolean

MyJavaClass.fun2("hi there")
// class java.lang.String

MyJavaClass.fun2(new Number(23));
// class jdk.nashorn.internal.objects.NativeNumber

MyJavaClass.fun2(new Date());
// class jdk.nashorn.internal.objects.NativeDate

MyJavaClass.fun2(new RegExp());
// class jdk.nashorn.internal.objects.NativeRegExp

MyJavaClass.fun2({foo: 'bar'});
// class jdk.nashorn.internal.scripts.JO4
```

Primitive javascript types are converted to the appropriate java wrapper class. Instead native javascript objects are represented by internal adapter classes. Please keep in mind that classes from `jdk.nashorn.internal` are subject to change, so you shouldn't program against those classes in client-code:

> Anything marked internal will likely change out from underneath you.

ScriptObjectMirror

When passing native javascript objects to java you can utilize the class `ScriptObjectMirror` which is actually a java representation of the underlying javascript object. ScriptObjectMirror implements the map interface and resides inside the package `jdk.nashorn.api`. Classes from this package are intended to be used in client-code.

The next sample changes the parameter type from `Object` to `ScriptObjectMirror` so we can extract some infos from the passed javascript object:

```
static void fun3(ScriptObjectMirror mirror) {
    System.out.println(mirror.getClassName() + ": " +
        Arrays.toString(mirror.getOwnKeys(true)));
}
```

When passing an object hash to this method, the properties are accessible on the java side:

```
MyJavaClass.fun3({
    foo: 'bar',
    bar: 'foo'
});

// Object: [foo, bar]
```

We can also call member functions on javascript object from java. Let's first define a javascript type Person with properties `firstName` and `lastName` and method `getFullName`.

```
function Person(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.getFullName = function() {
        return this.firstName + " " + this.lastName;
    }
}
```

The javascript method `getFullName` can be called on the ScriptObjectMirror via `callMember()`.

```
static void fun4(ScriptObjectMirror person) {
    System.out.println("Full Name is: " + person.callMember("getFullName"));
}
```

When passing a new person to the java method, we see the desired result on the console:

```
var person1 = new Person("Peter", "Parker");
MyJavaClass.fun4(person1);

// Full Name is: Peter Parker
```

Language Extensions

Nashorn defines various language and API extensions to the ECMAScript standard. Let's head right into the most recent features:

Typed Arrays

Native javascript arrays are untyped. Nashorn enables you to use typed java arrays in javascript:

```
var IntArray = Java.type("int[]");

var array = new IntArray(5);
array[0] = 5;
array[1] = 4;
array[2] = 3;
array[3] = 2;
array[4] = 1;

try {
    array[5] = 23;
} catch (e) {
    print(e.message);  // Array index out of range: 5
}

array[0] = "17";
print(array[0]);  // 17

array[0] = "wrong type";
print(array[0]);  // 0

array[0] = "17.3";
print(array[0]);  // 17
```

The `int[]` array behaves like a real java int array. But additionally Nashorn performs implicit type conversions under the hood when we're trying to add non-integer values to the array. Strings will be auto-converted to int which is quite handy.

Collections and For Each

Instead of messing around with arrays we can use any java collection. First define the java type via `Java.type`, then create new instances on demand.

```
var ArrayList = Java.type('java.util.ArrayList');
var list = new ArrayList();
list.add('a');
list.add('b');
list.add('c');

for each (var el in list) print(el);  // a, b, c
```

In order to iterate over collections and arrays Nashorn introduces the `for each` statement. It works just like the foreach loop in java.

Here's another collection foreach example, utilizing `HashMap`:

```
var map = new java.util.HashMap();
map.put('foo', 'val1');
map.put('bar', 'val2');

for each (var e in map.keySet()) print(e);  // foo, bar

for each (var e in map.values()) print(e);  // val1, val2
```

Lambda expressions and Streams

Everyone loves lambdas and streams - so does Nashorn! Although ECMAScript 5.1 lacks the compact arrow syntax from the Java 8 lambda expressions, we can use function literals where ever lambda expressions are accepted.

```
var list2 = new java.util.ArrayList();
list2.add("ddd2");
list2.add("aaa2");
list2.add("bbb1");
list2.add("aaa1");
list2.add("bbb3");
list2.add("ccc");
list2.add("bbb2");
list2.add("ddd1");

list2
    .stream()
    .filter(function(el) {
        return el.startsWith("aaa");
    })
    .sorted()
    .forEach(function(el) {
        print(el);
    });
    // aaa1, aaa2
```

Extending classes

Java types can simply be extended with the `Java.extend` extension. As you can see in the next example, you can even create multi-threaded code in your scripts:

```
var Runnable = Java.type('java.lang.Runnable');
var Printer = Java.extend(Runnable, {
    run: function() {
        print('printed from a separate thread');
    }
});

var Thread = Java.type('java.lang.Thread');
new Thread(new Printer()).start();

new Thread(function() {
    print('printed from another thread');
}).start();

// printed from a separate thread
// printed from another thread
```

Parameter overloading

Methods and functions can either be called with the point notation or with the square braces notation.

```
var System = Java.type('java.lang.System');
System.out.println(10);              // 10
System.out["println"](11.0);         // 11.0
System.out["println(double)"](12);   // 12.0
```

Passing the optional parameter type `println(double)` when calling a method with overloaded parameters determines the exact method to be called.

Java Beans

Instead of explicitly working with getters and setters you can just use simple property names both for getting or setting values from a java bean.

```
var Date = Java.type('java.util.Date');
var date = new Date();
date.year += 1900;
print(date.year);  // 2014
```

Function Literals

For simple one line functions we can skip the curly braces:

```
function sqr(x) x * x;
print(sqr(3));    // 9
```

Binding properties

Properties from two different objects can be bound together:

```
var o1 = {};
var o2 = { foo: 'bar'};

Object.bindProperties(o1, o2);

print(o1.foo);    // bar
o1.foo = 'BAM';
print(o2.foo);    // BAM
```

Trimming strings

I like my strings trimmed.

```
print("   hehe".trimLeft());            // hehe
print("hehe    ".trimRight() + "he");   // hehehe
```

Whereis

In case you forget where you are:

```
print(__FILE__, __LINE__, __DIR__);
```

Import Scopes

Sometimes it's useful to import many java packages at once. We can use the class `JavaImporter` to be used in conjunction with the `with` statement. All class files from the imported packages are accessible within the local scope of the `with` statement:

```
var imports = new JavaImporter(java.io, java.lang);
with (imports) {
    var file = new File(__FILE__);
    System.out.println(file.getAbsolutePath());
    // /path/to/my/script.js
}
```

Convert arrays

Some packages like `java.util` can be accessed directly without utilizing `Java.type` or `JavaImporter`:

```
var list = new java.util.ArrayList();
list.add("s1");
list.add("s2");
list.add("s3");
```

This code converts the java list to a native javascript array:

```
var jsArray = Java.from(list);
print(jsArray);                                  // s1,s2,s3
print(Object.prototype.toString.call(jsArray));  // [object Array]
```

And the other way around:

```
var javaArray = Java.to([3, 5, 7, 11], "int[]");
```

Calling Super

Accessing overridden members in javascript is traditionally awkward because javas `super` keyword doesn't exist in ECMAScript. Luckily nashorn goes to the rescue.

First we define a super type in java code:

```
class SuperRunner implements Runnable {
    @Override
    public void run() {
        System.out.println("super run");
    }
}
```

Next we override `SuperRunner` from javascript. Pay attention to the extended nashorn syntax when creating a new `Runner` instance: The syntax of overriding members is borrowed from javas anonymous objects.

```
var SuperRunner = Java.type('com.winterbe.java8.SuperRunner');
var Runner = Java.extend(SuperRunner);

var runner = new Runner() {
    run: function() {
        Java.super(runner).run();
        print('on my run');
    }
}
runner.run();

// super run
// on my run
```

We call the overridden method `SuperRunner.run()` by utilizing the `Java.super` extension.

Loading scripts

Evaluating additional script files from javascript is quite easy. We can load both local or remote scripts with the `load` function.

I'm using Underscore.js a lot for my web front-ends, so let's reuse Underscore in Nashorn:

```
load('http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.6.0/underscore-min.js');

var odds = _.filter([1, 2, 3, 4, 5, 6], function (num) {
    return num % 2 == 1;
});

print(odds);  // 1, 3, 5
```

The external script will be evaluated in the same javascript context, so we can access the underscore variable directly. Keep in mind that loading scripts can potentially break your own code when variable names are overlapping each other.

This problem can be bypassed by loading script files into a new global context:

```
loadWithNewGlobal('script.js');
```

Command-line scripts

If you're interested in writing command-line (shell) scripts with Java, give Nake a try. Nake is a simplified Make for Java 8 Nashorn. You define tasks in a project-specific `Nakefile`, then run those tasks by typing `nake -- myTask` into the command line. Tasks are written in javascript and run in Nashorns scripting mode, so you can utilize the full power of your terminal as well as the JDK8 API and any java library.

For Java Developers writing command-line scripts is easy as never before...

That's it

I hope this guide was helpful to you and you enjoyed our journey to the Nashorn Javascript Engine. For further information about Nashorn read here, here and here. A guide to coding shell scripts with Nashorn can be found here.

I recently published a follow up article about how to use Backbone.js models with the Nashorn Javascript Engine. If you want to learn more about Java 8 feel free to read my Java 8 Tutorial and my Java 8 Stream Tutorial.

The runnable source code from this Nashorn tutorial is hosted on GitHub. Feel free to fork the repository or send me your feedback via Twitter.

Keep on coding!

Java 8 Concurrency Tutorial: Threads and Executors

April 07, 2015

Welcome to the first part of my Java 8 Concurrency tutorial. This guide teaches you concurrent programming in Java 8 with easily understood code examples. It's the first part out of a series of tutorials covering the Java Concurrency API. In the next 15 min you learn how to execute code in parallel via threads, tasks and executor services.

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

The Concurrency API was first introduced with the release of Java 5 and then progressively enhanced with every new Java release. The majority of concepts shown in this article also work in older versions of Java. However my code samples focus on Java 8 and make heavy use of lambda expressions and other new features. If you're not yet familiar with lambdas I recommend reading my Java 8 Tutorial first.

Threads and Runnables

All modern operating systems support concurrency both via processes) and threads. Processes are instances of programs which typically run independent to each other, e.g. if you start a java program the operating system spawns a new process which runs in parallel to other programs. Inside those processes we can utilize threads to execute code concurrently, so we can make the most out of the available cores of the CPU.

Java supports Threads since JDK 1.0. Before starting a new thread you have to specify the code to be executed by this thread, often called the *task*. This is done by implementing `Runnable` - a functional interface defining a single void no-args method `run()` as demonstrated in the following example:

```
Runnable task = () -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
};

task.run();

Thread thread = new Thread(task);
thread.start();

System.out.println("Done!");
```

Since `Runnable` is a functional interface we can utilize Java 8 lambda expressions to print the current threads name to the console. First we execute the runnable directly on the main thread before starting a new thread.

The result on the console might look like this:

```
Hello main
Hello Thread-0
Done!
```

Or that:

```
Hello main
Done!
Hello Thread-0
```

Due to concurrent execution we cannot predict if the runnable will be invoked before or after printing 'done'. The order is non-deterministic, thus making concurrent programming a complex task in larger applications.

Threads can be put to sleep for a certain duration. This is quite handy to simulate long running tasks in the subsequent code samples of this article:

```
Runnable runnable = () -> {
    try {
        String name = Thread.currentThread().getName();
        System.out.println("Foo " + name);
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Bar " + name);
    }
    catch (InterruptedException e) {
        e.printStackTrace();
    }
};

Thread thread = new Thread(runnable);
thread.start();
```

When you run the above code you'll notice the one second delay between the first and the second print statement. `TimeUnit` is a useful enum for working with units of time. Alternatively you can achieve the same by calling `Thread.sleep(1000)`.

Working with the `Thread` class can be very tedious and error-prone. Due to that reason the **Concurrency API** has been introduced back in 2004 with the release of Java 5. The API is located in package `java.util.concurrent` and contains many useful classes for handling concurrent programming. Since that time the Concurrency API has been enhanced with every new Java release and even Java 8 provides new classes and methods for dealing with concurrency.

Now let's take a deeper look at one of the most important parts of the Concurrency API - the executor services.

Executors

The Concurrency API introduces the concept of an `ExecutorService` as a higher level replacement for working with threads directly. Executors are capable of running asynchronous tasks and typically manage a pool of threads, so we don't have to create new threads manually. All threads of the internal pool will be reused under the hood for revenant tasks, so we can run as many concurrent tasks as we want throughout the life-cycle of our application with a single executor service.

This is how the first thread-example looks like using executors:

```
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);
});

// => Hello pool-1-thread-1
```

The class `Executors` provides convenient factory methods for creating different kinds of executor services. In this sample we use an executor with a thread pool of size one.

The result looks similar to the above sample but when running the code you'll notice an important difference: the java process never stops! Executors have to be stopped explicitly - otherwise they keep listening for new tasks.

An `ExecutorService` provides two methods for that purpose: `shutdown()` waits for currently running tasks to finish while `shutdownNow()` interrupts all running tasks and shut the executor down immediately.

This is the preferred way how I typically shutdown executors:

```
try {
    System.out.println("attempt to shutdown executor");
    executor.shutdown();
    executor.awaitTermination(5, TimeUnit.SECONDS);
}
catch (InterruptedException e) {
    System.err.println("tasks interrupted");
}
finally {
    if (!executor.isTerminated()) {
        System.err.println("cancel non-finished tasks");
    }
    executor.shutdownNow();
    System.out.println("shutdown finished");
}
```

The executor shuts down softly by waiting a certain amount of time for termination of currently running tasks. After a maximum of five seconds the executor finally shuts down by interrupting all running tasks.

Callables and Futures

In addition to `Runnable` executors support another kind of task named `Callable`. Callables are functional interfaces just like runnables but instead of being `void` they return a value.

This lambda expression defines a callable returning an integer after sleeping for one second:

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Callables can be submitted to executor services just like runnables. But what about the callables result? Since `submit()` doesn't wait until the task completes, the executor service cannot return the result of the callable directly. Instead the executor returns a special result of type `Future` which can be used to retrieve the actual result at a later point in time.

```
ExecutorService executor = Executors.newFixedThreadPool(1);
Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());
System.out.print("result: " + result);
```

After submitting the callable to the executor we first check if the future has already been finished execution via `isDone()`. I'm pretty sure this isn't the case since the above callable sleeps for one second before returning the integer.

Calling the method `get()` blocks the current thread and waits until the callable completes before returning the actual result `123`. Now the future is finally done and we see the following result on the console:

```
future done? false
future done? true
result: 123
```

Futures are tightly coupled to the underlying executor service. Keep in mind that every non-terminated future will throw exceptions if you shutdown the executor:

```
executor.shutdownNow();
future.get();
```

You might have noticed that the creation of the executor slightly differs from the previous example. We use `newFixedThreadPool(1)` to create an executor service backed by a thread-pool of size one. This is equivalent to `newSingleThreadExecutor()` but we could later increase the pool size by simply passing a value larger than one.

Timeouts

Any call to `future.get()` will block and wait until the underlying callable has been terminated. In the worst case a callable runs forever - thus making your application unresponsive. You can simply counteract those scenarios by passing a timeout:

```
ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(() -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
});

future.get(1, TimeUnit.SECONDS);
```

Executing the above code results in a `TimeoutException`:

```
Exception in thread "main" java.util.concurrent.TimeoutException
    at java.util.concurrent.FutureTask.get(FutureTask.java:205)
```

You might already have guessed why this exception is thrown: We specified a maximum wait time of one second but the callable actually needs two seconds before returning the result.

InvokeAll

Executors support batch submitting of multiple callables at once via `invokeAll()`. This method accepts a collection of callables and returns a list of futures.

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");

executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

In this example we utilize Java 8 functional streams in order to process all futures returned by the invocation of `invokeAll`. We first map each future to its return value and then print each value to the console. If you're not yet familiar with streams read my Java 8 Stream Tutorial.

InvokeAny

Another way of batch-submitting callables is the method `invokeAny()` which works slightly different to `invokeAll()`. Instead of returning future objects this method blocks until the first callable terminates and returns the result of that callable.

In order to test this behavior we use this helper method to simulate callables with different durations. The method returns a callable that sleeps for a certain amount of time until returning the given result:

```
Callable<String> callable(String result, long sleepSeconds) {
    return () -> {
        TimeUnit.SECONDS.sleep(sleepSeconds);
        return result;
    };
}
```

We use this method to create a bunch of callables with different durations from one to three seconds. Submitting those callables to an executor via `invokeAny()` returns the string result of the fastest callable - in that case task2:

```
ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
    callable("task1", 2),
    callable("task2", 1),
    callable("task3", 3));

String result = executor.invokeAny(callables);
System.out.println(result);

// => task2
```

The above example uses yet another type of executor created via `newWorkStealingPool()`. This factory method is part of Java 8 and returns an executor of type `ForkJoinPool` which works slightly different than normal executors. Instead of using a fixed size thread-pool ForkJoinPools are created for a given parallelism size which per default is the number of available cores of the hosts CPU.

ForkJoinPools exist since Java 7 and will be covered in detail in a later tutorial of this series. Let's finish this tutorial by taking a deeper look at scheduled executors.

Scheduled Executors

We've already learned how to submit and run tasks once on an executor. In order to periodically run common tasks multiple times, we can utilize scheduled thread pools.

A `ScheduledExecutorService` is capable of scheduling tasks to run either periodically or once after a certain amount of time has elapsed.

This code sample schedules a task to run after an initial delay of three seconds has passed:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());
ScheduledFuture<?> future = executor.schedule(task, 3, TimeUnit.SECONDS);

TimeUnit.MILLISECONDS.sleep(1337);

long remainingDelay = future.getDelay(TimeUnit.MILLISECONDS);
System.out.printf("Remaining Delay: %sms", remainingDelay);
```

Scheduling a task produces a specialized future of type `ScheduledFuture` which - in addition to `Future` - provides the method `getDelay()` to retrieve the remaining delay. After this delay has elapsed the task will be executed concurrently.

In order to schedule tasks to be executed periodically, executors provide the two methods `scheduleAtFixedRate()` and `scheduleWithFixedDelay()`. The first method is capable of executing tasks with a fixed time rate, e.g. once every second as demonstrated in this example:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> System.out.println("Scheduling: " + System.nanoTime());

int initialDelay = 0;
int period = 1;
executor.scheduleAtFixedRate(task, initialDelay, period, TimeUnit.SECONDS);
```

Additionally this method accepts an initial delay which describes the leading wait time before the task will be executed for the first time.

Please keep in mind that `scheduleAtFixedRate()` doesn't take into account the actual duration of the task. So if you specify a period of one second but the task needs 2 seconds to be executed then the thread pool will working to capacity very soon.

In that case you should consider using `scheduleWithFixedDelay()` instead. This method works just like the counterpart described above. The difference is that the wait time period applies between the end of a task and the start of the next task. For example:

```
ScheduledExecutorService executor = Executors.newScheduledThreadPool(1);

Runnable task = () -> {
    try {
        TimeUnit.SECONDS.sleep(2);
        System.out.println("Scheduling: " + System.nanoTime());
    }
    catch (InterruptedException e) {
        System.err.println("task interrupted");
    }
};

executor.scheduleWithFixedDelay(task, 0, 1, TimeUnit.SECONDS);
```

This example schedules a task with a fixed delay of one second between the end of an execution and the start of the next execution. The initial delay is zero and the tasks duration is two seconds. So we end up with an execution interval of 0s, 3s, 6s, 9s and so on. As you can see `scheduleWithFixedDelay()` is handy if you cannot predict the duration of the scheduled tasks.

This was the first part out of a series of concurrency tutorials. I recommend practicing the shown code samples by your own. You find all code samples from this article on GitHub, so feel free to fork the repo and give me star.

I hope you've enjoyed this article. If you have any further questions send me your feedback in the comments below or via Twitter.

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

Java 8 Concurrency Tutorial: Synchronization and Locks

April 30, 2015

Welcome to the second part of my Java 8 Concurrency Tutorial out of a series of guides teaching multi-threaded programming in Java 8 with easily understood code examples. In the next 15 min you learn how to synchronize access to mutable shared variables via the synchronized keyword, locks and semaphores.

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

The majority of concepts shown in this article also work in older versions of Java. However the code samples focus on Java 8 and make heavy use of lambda expressions and new concurrency features. If you're not yet familiar with lambdas I recommend reading my Java 8 Tutorial first.

For simplicity the code samples of this tutorial make use of the two helper methods `sleep(seconds)` and `stop(executor)` as defined here.

Synchronized

In the previous tutorial) we've learned how to execute code in parallel via executor services. When writing such multi-threaded code you have to pay particular attention when accessing shared mutable variables concurrently from multiple threads. Let's just say we want to increment an integer which is accessible simultaneously from multiple threads.

We define a field `count` with a method `increment()` to increase count by one:

```
int count = 0;

void increment() {
    count = count + 1;
}
```

When calling this method concurrently from multiple threads we're in serious trouble:

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)
    .forEach(i -> executor.submit(this::increment));

stop(executor);

System.out.println(count);  // 9965
```

Instead of seeing a constant result count of 10000 the actual result varies with every execution of the above code. The reason is that we share a mutable variable upon different threads without synchronizing the access to this variable which results in a race condition.

Three steps have to be performed in order to increment the number: (i) read the current value, (ii) increase this value by one and (iii) write the new value to the variable. If two threads perform these steps in parallel it's possible that both threads perform step 1 simultaneously thus reading the same current value. This results in lost writes so the actual result is lower. In the above sample 35 increments got lost due to concurrent unsynchronized access to count but you may see different results when executing the code by yourself.

Luckily Java supports thread-synchronization since the early days via the `synchronized` keyword. We can utilize `synchronized` to fix the above race conditions when incrementing the count:

```
synchronized void incrementSync() {
    count = count + 1;
}
```

When using `incrementSync()` concurrently we get the desired result count of 10000. No race conditions occur any longer and the result is stable with every execution of the code:

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10000)
    .forEach(i -> executor.submit(this::incrementSync));

stop(executor);

System.out.println(count);  // 10000
```

The `synchronized` keyword is also available as a block statement.

```
void incrementSync() {
    synchronized (this) {
        count = count + 1;
    }
}
```

Internally Java uses a so called *monitor* also known as monitor lock or intrinsic lock in order to manage synchronization. This monitor is bound to an object, e.g. when using synchronized methods each method share the same monitor of the corresponding object.

All implicit monitors implement the *reentrant* characteristics. Reentrant means that locks are bound to the current thread. A thread can safely acquire the same lock multiple times without running into deadlocks (e.g. a synchronized method calls another synchronized method on the same object).

Locks

Instead of using implicit locking via the `synchronized` keyword the Concurrency API supports various explicit locks specified by the `Lock` interface. Locks support various methods for finer grained lock control thus are more expressive than implicit monitors.

Multiple lock implementations are available in the standard JDK which will be demonstrated in the following sections.

ReentrantLock

The class `ReentrantLock` is a mutual exclusion lock with the same basic behavior as the implicit monitors accessed via the `synchronized` keyword but with extended capabilities. As the name suggests this lock implements reentrant characteristics just as implicit monitors.

Let's see how the above sample looks like using `ReentrantLock` :

```
ReentrantLock lock = new ReentrantLock();
int count = 0;

void increment() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();
    }
}
```

A lock is acquired via `lock()` and released via `unlock()` . It's important to wrap your code into a `try/finally` block to ensure unlocking in case of exceptions. This method is thread-safe just like the synchronized counterpart. If another thread has already acquired the lock subsequent calls to `lock()` pause the current thread until the lock has been unlocked. Only one thread can hold the lock at any given time.

Locks support various methods for fine grained control as seen in the next sample:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
ReentrantLock lock = new ReentrantLock();

executor.submit(() -> {
    lock.lock();
    try {
        sleep(1);
    } finally {
        lock.unlock();
    }
});

executor.submit(() -> {
    System.out.println("Locked: " + lock.isLocked());
    System.out.println("Held by me: " + lock.isHeldByCurrentThread());
    boolean locked = lock.tryLock();
    System.out.println("Lock acquired: " + locked);
});

stop(executor);
```

While the first task holds the lock for one second the second task obtains different information about the current state of the lock:

```
Locked: true
Held by me: false
Lock acquired: false
```

The method `tryLock()` as an alternative to `lock()` tries to acquire the lock without pausing the current thread. The boolean result must be used to check if the lock has actually been acquired before accessing any shared mutable variables.

ReadWriteLock

The interface `ReadWriteLock` specifies another type of lock maintaining a pair of locks for read and write access. The idea behind read-write locks is that it's usually safe to read mutable variables concurrently as long as nobody is writing to this variable. So the read-lock can be held simultaneously by multiple threads as long as no threads hold the write-lock. This can improve performance and throughput in case that reads are more frequent than writes.

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
ReadWriteLock lock = new ReentrantReadWriteLock();

executor.submit(() -> {
    lock.writeLock().lock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.writeLock().unlock();
    }
});
```

The above example first acquires a write-lock in order to put a new value to the map after sleeping for one second. Before this task has finished two other tasks are being submitted trying to read the entry from the map and sleep for one second:

```
Runnable readTask = () -> {
    lock.readLock().lock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.readLock().unlock();
    }
};

executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

When you execute this code sample you'll notice that both read tasks have to wait the whole second until the write task has finished. After the write lock has been released both read tasks are executed in parallel and print the result simultaneously to the console. They don't have to wait for each other to finish because read-locks can safely be acquired concurrently as long as no write-lock is held by another thread.

StampedLock

Java 8 ships with a new kind of lock called `StampedLock` which also support read and write locks just like in the example above. In contrast to `ReadWriteLock` the locking methods of a `StampedLock` return a stamp represented by a `long` value. You can use these stamps to either release a lock or to check if the lock is still valid. Additionally stamped locks support another lock mode called *optimistic locking*.

Let's rewrite the last example code to use `StampedLock` instead of `ReadWriteLock`:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
Map<String, String> map = new HashMap<>();
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        sleep(1);
        map.put("foo", "bar");
    } finally {
        lock.unlockWrite(stamp);
    }
});

Runnable readTask = () -> {
    long stamp = lock.readLock();
    try {
        System.out.println(map.get("foo"));
        sleep(1);
    } finally {
        lock.unlockRead(stamp);
    }
};

executor.submit(readTask);
executor.submit(readTask);

stop(executor);
```

Obtaining a read or write lock via `readLock()` or `writeLock()` returns a stamp which is later used for unlocking within the finally block. Keep in mind that stamped locks don't implement reentrant characteristics. Each call to lock returns a new stamp and blocks if no lock is available even if the same thread already holds a lock. So you have to pay particular attention not to run into deadlocks.

Just like in the previous `ReadWriteLock` example both read tasks have to wait until the write lock has been released. Then both read tasks print to the console simultaneously because multiple reads doesn't block each other as long as no write-lock is held.

The next example demonstrates *optimistic locking*:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.tryOptimisticRead();
    try {
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        sleep(1);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
        sleep(2);
        System.out.println("Optimistic Lock Valid: " + lock.validate(stamp));
    } finally {
        lock.unlock(stamp);
    }
});

executor.submit(() -> {
    long stamp = lock.writeLock();
    try {
        System.out.println("Write Lock acquired");
        sleep(2);
    } finally {
        lock.unlock(stamp);
        System.out.println("Write done");
    }
});

stop(executor);
```

An optimistic read lock is acquired by calling `tryOptimisticRead()` which always returns a stamp without blocking the current thread, no matter if the lock is actually available. If there's already a write lock active the returned stamp equals zero. You can always check if a stamp is valid by calling `lock.validate(stamp)`.

Executing the above code results in the following output:

```
Optimistic Lock Valid: true
Write Lock acquired
Optimistic Lock Valid: false
Write done
Optimistic Lock Valid: false
```

The optimistic lock is valid right after acquiring the lock. In contrast to normal read locks an optimistic lock doesn't prevent other threads to obtain a write lock instantaneously. After sending the first thread to sleep for one second the second thread obtains a write lock without waiting for the optimistic read lock to be released. From this point the optimistic read lock is no longer valid. Even when the write lock is released the optimistic read locks stays invalid.

So when working with optimistic locks you have to validate the lock every time *after* accessing any shared mutable variable to make sure the read was still valid.

Sometimes it's useful to convert a read lock into a write lock without unlocking and locking again. `StampedLock` provides the method `tryConvertToWriteLock()` for that purpose as seen in the next sample:

```
ExecutorService executor = Executors.newFixedThreadPool(2);
StampedLock lock = new StampedLock();

executor.submit(() -> {
    long stamp = lock.readLock();
    try {
        if (count == 0) {
            stamp = lock.tryConvertToWriteLock(stamp);
            if (stamp == 0L) {
                System.out.println("Could not convert to write lock");
                stamp = lock.writeLock();
            }
            count = 23;
        }
        System.out.println(count);
    } finally {
        lock.unlock(stamp);
    }
});

stop(executor);
```

The task first obtains a read lock and prints the current value of field `count` to the console. But if the current value is zero we want to assign a new value of `23`. We first have to convert the read lock into a write lock to not break potential concurrent access by other threads. Calling `tryConvertToWriteLock()` doesn't block but may return a zero stamp indicating that no write lock is currently available. In that case we call `writeLock()` to block the current thread until a write lock is available.

Semaphores

In addition to locks the Concurrency API also supports counting semaphores. Whereas locks usually grant exclusive access to variables or resources, a semaphore is capable of maintaining whole sets of permits. This is useful in different scenarios where you have to limit the amount concurrent access to certain parts of your application.

Here's an example how to limit access to a long running task simulated by `sleep(5)`:

```
ExecutorService executor = Executors.newFixedThreadPool(10);

Semaphore semaphore = new Semaphore(5);

Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            sleep(5);
        } else {
            System.out.println("Could not acquire semaphore");
        }
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {
            semaphore.release();
        }
    }
}

IntStream.range(0, 10)
    .forEach(i -> executor.submit(longRunningTask));

stop(executor);
```

The executor can potentially run 10 tasks concurrently but we use a semaphore of size 5, thus limiting concurrent access to 5. It's important to use a `try/finally` block to properly release the semaphore even in case of exceptions.

Executing the above code results in the following output:

```
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Semaphore acquired
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
Could not acquire semaphore
```

The semaphores permits access to the actual long running operation simulated by `sleep(5)` up to a maximum of 5. Every subsequent call to `tryAcquire()` elapses the maximum wait timeout of one second, resulting in the appropriate console output that no semaphore could be acquired.

This was the second part out of a series of concurrency tutorials. More parts will be released in the near future, so stay tuned. As usual you find all code samples from this article on GitHub, so feel free to fork the repo and try it by your own.

I hope you've enjoyed this article. If you have any further questions send me your feedback in the comments below. You should also follow me on Twitter for more dev-related stuff!

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

Java 8 Concurrency Tutorial: Atomic Variables and ConcurrentMap

May 22, 2015

Welcome to the third part of my tutorial series about multi-threaded programming in Java 8. This tutorial covers two important parts of the Concurrency API: Atomic Variables and Concurrent Maps. Both have been greatly improved with the introduction of lambda expressions and functional programming in the latest Java 8 release. All those new features are described with a bunch of easily understood code samples. Enjoy!

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

For simplicity the code samples of this tutorial make use of the two helper methods `sleep(seconds)` and `stop(executor)` as defined here.

AtomicInteger

The package `java.concurrent.atomic` contains many useful classes to perform atomic operations. An operation is atomic when you can safely perform the operation in parallel on multiple threads without using the `synchronized` keyword or locks as shown in my previous tutorial.

Internally, the atomic classes make heavy use of compare-and-swap (CAS), an atomic instruction directly supported by most modern CPUs. Those instructions usually are much faster than synchronizing via locks. So my advice is to prefer atomic classes over locks in case you just have to change a single mutable variable concurrently.

Now let's pick one of the atomic classes for a few examples: `AtomicInteger`

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> executor.submit(atomicInt::incrementAndGet));

stop(executor);

System.out.println(atomicInt.get());    // => 1000
```

By using `AtomicInteger` as a replacement for `Integer` we're able to increment the number concurrently in a thread-safe manor without synchronizing the access to the variable. The method `incrementAndGet()` is an atomic operation so we can safely call this method from multiple threads.

AtomicInteger supports various kinds of atomic operations. The method `updateAndGet()` accepts a lambda expression in order to perform arbitrary arithmetic operations upon the integer:

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.updateAndGet(n -> n + 2);
        executor.submit(task);
    });

stop(executor);

System.out.println(atomicInt.get());    // => 2000
```

The method `accumulateAndGet()` accepts another kind of lambda expression of type `IntBinaryOperator`. We use this method to sum up all values from 0 to 1000 concurrently in the next sample:

```
AtomicInteger atomicInt = new AtomicInteger(0);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.accumulateAndGet(i, (n, m) -> n + m);
        executor.submit(task);
    });

stop(executor);

System.out.println(atomicInt.get());    // => 499500
```

Other useful atomic classes are AtomicBoolean, AtomicLong and AtomicReference.

LongAdder

The class `LongAdder` as an alternative to `AtomicLong` can be used to consecutively add values to a number.

```
ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 1000)
    .forEach(i -> executor.submit(adder::increment));

stop(executor);

System.out.println(adder.sumThenReset());   // => 1000
```

LongAdder provides methods `add()` and `increment()` just like the atomic number classes and is also thread-safe. But instead of summing up a single result this class maintains a set of variables internally to reduce contention over threads. The actual result can be retrieved by calling `sum()` or `sumThenReset()`.

This class is usually preferable over atomic numbers when updates from multiple threads are more common than reads. This is often the case when capturing statistical data, e.g. you want to count the number of requests served on a web server. The drawback of `LongAdder` is higher memory consumption because a set of variables is held in-memory.

LongAccumulator

LongAccumulator is a more generalized version of LongAdder. Instead of performing simple add operations the class `LongAccumulator` builds around a lambda expression of type `LongBinaryOperator` as demonstrated in this code sample:

```
LongBinaryOperator op = (x, y) -> 2 * x + y;
LongAccumulator accumulator = new LongAccumulator(op, 1L);

ExecutorService executor = Executors.newFixedThreadPool(2);

IntStream.range(0, 10)
    .forEach(i -> executor.submit(() -> accumulator.accumulate(i)));

stop(executor);

System.out.println(accumulator.getThenReset());     // => 2539
```

We create a LongAccumulator with the function `2 * x + y` and an initial value of one. With every call to `accumulate(i)` both the current result and the value `i` are passed as parameters to the lambda expression.

A `LongAccumulator` just like `LongAdder` maintains a set of variables internally to reduce contention over threads.

ConcurrentMap

The interface `ConcurrentMap` extends the map interface and defines one of the most useful concurrent collection types. Java 8 introduces functional programming by adding new methods to this interface.

In the next code snippets we use the following sample map to demonstrates those new methods:

```
ConcurrentMap<String, String> map = new ConcurrentHashMap<>();
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

The method `forEach()` accepts a lambda expression of type `BiConsumer` with both the key and value of the map passed as parameters. It can be used as a replacement to for-each loops to iterate over the entries of the concurrent map. The iteration is performed sequentially on the current thread.

```
map.forEach((key, value) -> System.out.printf("%s = %s\n", key, value));
```

The method `putIfAbsent()` puts a new value into the map only if no value exists for the given key. At least for the `ConcurrentHashMap` implementation of this method is thread-safe just like `put()` so you don't have to synchronize when accessing the map concurrently from different threads:

```
String value = map.putIfAbsent("c3", "p1");
System.out.println(value);    // p0
```

The method `getOrDefault()` returns the value for the given key. In case no entry exists for this key the passed default value is returned:

```
String value = map.getOrDefault("hi", "there");
System.out.println(value);    // there
```

The method `replaceAll()` accepts a lambda expression of type `BiFunction`. BiFunctions take two parameters and return a single value. In this case the function is called with the key and the value of each map entry and returns a new value to be assigned for the current key:

```
map.replaceAll((key, value) -> "r2".equals(key) ? "d3" : value);
System.out.println(map.get("r2"));    // d3
```

Instead of replacing all values of the map `compute()` let's us transform a single entry. The method accepts both the key to be computed and a bi-function to specify the transformation of the value.

```
map.compute("foo", (key, value) -> value + value);
System.out.println(map.get("foo"));   // barbar
```

In addition to `compute()` two variants exist: `computeIfAbsent()` and `computeIfPresent()`. The functional parameters of these methods only get called if the key is absent or present respectively.

Finally, the method `merge()` can be utilized to unify a new value with an existing value in the map. Merge accepts a key, the new value to be merged into the existing entry and a bi-function to specify the merging behavior of both values:

```
map.merge("foo", "boo", (oldVal, newVal) -> newVal + " was " + oldVal);
System.out.println(map.get("foo"));   // boo was foo
```

ConcurrentHashMap

All those methods above are part of the `ConcurrentMap` interface, thereby available to all implementations of that interface. In addition the most important implementation `ConcurrentHashMap` has been further enhanced with a couple of new methods to perform parallel operations upon the map.

Just like parallel streams those methods use a special `ForkJoinPool` available via `ForkJoinPool.commonPool()` in Java 8. This pool uses a preset parallelism which depends on the number of available cores. Four CPU cores are available on my machine which results in a parallelism of three:

```
System.out.println(ForkJoinPool.getCommonPoolParallelism());  // 3
```

This value can be decreased or increased by setting the following JVM parameter:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

We use the same example map for demonstrating purposes but this time we work upon the concrete implementation `ConcurrentHashMap` instead of the interface `ConcurrentMap`, so we can access all public methods from this class:

```
ConcurrentHashMap<String, String> map = new ConcurrentHashMap<>();
map.put("foo", "bar");
map.put("han", "solo");
map.put("r2", "d2");
map.put("c3", "p0");
```

Java 8 introduces three kinds of parallel operations: `forEach`, `search` and `reduce`. Each of those operations are available in four forms accepting functions with keys, values, entries and key-value pair arguments.

All of those methods use a common first argument called `parallelismThreshold`. This threshold indicates the minimum collection size when the operation should be executed in parallel. E.g. if you pass a threshold of 500 and the actual size of the map is 499 the operation will be performed sequentially on a single thread. In the next examples we use a threshold of one to always force parallel execution for demonstrating purposes.

ForEach

The method `forEach()` is capable of iterating over the key-value pairs of the map in parallel. The lambda expression of type `BiConsumer` is called with the key and value of the current iteration step. In order to visualize parallel execution we print the current threads name to the console. Keep in mind that in my case the underlying `ForkJoinPool` uses up to a maximum of three threads.

```
map.forEach(1, (key, value) ->
    System.out.printf("key: %s; value: %s; thread: %s\n",
        key, value, Thread.currentThread().getName()));

// key: r2; value: d2; thread: main
// key: foo; value: bar; thread: ForkJoinPool.commonPool-worker-1
// key: han; value: solo; thread: ForkJoinPool.commonPool-worker-2
// key: c3; value: p0; thread: main
```

Search

The method `search()` accepts a `BiFunction` returning a non-null search result for the current key-value pair or `null` if the current iteration doesn't match the desired search criteria. As soon as a non-null result is returned further processing is suppressed. Keep in mind that `ConcurrentHashMap` is unordered. The search function should not depend on the actual processing order of the map. If multiple entries of the map match the given search function the result may be non-deterministic.

```
String result = map.search(1, (key, value) -> {
    System.out.println(Thread.currentThread().getName());
    if ("foo".equals(key)) {
        return value;
    }
    return null;
});
System.out.println("Result: " + result);

// ForkJoinPool.commonPool-worker-2
// main
// ForkJoinPool.commonPool-worker-3
// Result: bar
```

Here's another example searching solely on the values of the map:

```
String result = map.searchValues(1, value -> {
    System.out.println(Thread.currentThread().getName());
    if (value.length() > 3) {
        return value;
    }
    return null;
});

System.out.println("Result: " + result);

// ForkJoinPool.commonPool-worker-2
// main
// main
// ForkJoinPool.commonPool-worker-1
// Result: solo
```

Reduce

The method `reduce()` already known from Java 8 Streams accepts two lambda expressions of type `BiFunction`. The first function transforms each key-value pair into a single value of any type. The second function combines all those transformed values into a single result, ignoring any possible `null` values.

```
String result = map.reduce(1,
    (key, value) -> {
        System.out.println("Transform: " + Thread.currentThread().getName());
        return key + "=" + value;
    },
    (s1, s2) -> {
        System.out.println("Reduce: " + Thread.currentThread().getName());
        return s1 + ", " + s2;
    });

System.out.println("Result: " + result);

// Transform: ForkJoinPool.commonPool-worker-2
// Transform: main
// Transform: ForkJoinPool.commonPool-worker-3
// Reduce: ForkJoinPool.commonPool-worker-3
// Transform: main
// Reduce: main
// Reduce: main
// Result: r2=d2, c3=p0, han=solo, foo=bar
```

I hope you've enjoyed reading the third part of my tutorial series about Java 8 Concurrency. The code samples from this tutorial are hosted on GitHub along with many other Java 8 code snippets. You're welcome to fork the repo and try it by your own.

If you want to support my work, please share this tutorial with your friends. You should also follow me on Twitter as I constantly tweet about Java and programming related stuff.

- Part 1: Threads and Executors
- Part 2: Synchronization and Locks
- Part 3: Atomic Variables and ConcurrentMap

Java 8 API by Example: Strings, Numbers, Math and Files

March 25, 2015

Plenty of tutorials and articles cover the most important changes in Java 8 like lambda expressions and functional streams. But furthermore many existing classes have been enhanced in the JDK 8 API with useful features and methods.

This article covers some of those smaller changes in the Java 8 API - each described with easily understood code samples. Let's take a deeper look into Strings, Numbers, Math and Files.

Slicing Strings

Two new methods are available on the String class: `join` and `chars`. The first method joins any number of strings into a single string with the given delimiter:

```
String.join(":", "foobar", "foo", "bar");
// => foobar:foo:bar
```

The second method `chars` creates a stream for all characters of the string, so you can use stream operations upon those characters:

```
"foobar:foo:bar"
    .chars()
    .distinct()
    .mapToObj(c -> String.valueOf((char)c))
    .sorted()
    .collect(Collectors.joining());
// => :abfor
```

Not only strings but also regex patterns now benefit from streams. Instead of splitting strings into streams for each character we can split strings for any pattern and create a stream to work upon as shown in this example:

```
Pattern.compile(":")
    .splitAsStream("foobar:foo:bar")
    .filter(s -> s.contains("bar"))
    .sorted()
    .collect(Collectors.joining(":"));
// => bar:foobar
```

Additionally regex patterns can be converted into predicates. Those predicates can for example be used to filter a stream of strings:

```
Pattern pattern = Pattern.compile(".*@gmail\\.com");
Stream.of("bob@gmail.com", "alice@hotmail.com")
    .filter(pattern.asPredicate())
    .count();
// => 1
```

The above pattern accepts any string which ends with `@gmail.com` and is then used as a Java 8 `Predicate` to filter a stream of email addresses.

Crunching Numbers

Java 8 adds additional support for working with unsigned numbers. Numbers in Java had always been signed. Let's look at `Integer` for example:

An `int` represents a maximum of 2^{32} binary digits. Numbers in Java are per default signed, so the last binary digit represents the sign (0 = positive, 1 = negative). Thus the maximum positive signed `int` is 2^{31} - 1 starting with the decimal zero.

You can access this value via `Integer.MAX_VALUE`:

```
System.out.println(Integer.MAX_VALUE);      // 2147483647
System.out.println(Integer.MAX_VALUE + 1);  // -2147483648
```

Java 8 adds support for parsing unsigned ints. Let's see how this works:

```
long maxUnsignedInt = (1l << 32) - 1;
String string = String.valueOf(maxUnsignedInt);
int unsignedInt = Integer.parseUnsignedInt(string, 10);
String string2 = Integer.toUnsignedString(unsignedInt, 10);
```

As you can see it's now possible to parse the maximum possible unsigned number 2^{32} - 1 into an integer. And you can also convert this number back into a string representing the unsigned number.

This wasn't possible before with `parseInt` as this example demonstrates:

```
try {
    Integer.parseInt(string, 10);
}
catch (NumberFormatException e) {
    System.err.println("could not parse signed int of " + maxUnsignedInt);
}
```

The number is not parseable as a signed int because it exceeds the maximum of 2^{31} - 1.

Do the Math

The utility class `Math` has been enhanced by a couple of new methods for handling number overflows. What does that mean? We've already seen that all number types have a maximum value. So what happens when the result of an arithmetic operation doesn't fit into its size?

```
System.out.println(Integer.MAX_VALUE);      // 2147483647
System.out.println(Integer.MAX_VALUE + 1);  // -2147483648
```

As you can see a so called **integer overflow** happens which is normally not the desired behavior.

Java 8 adds support for strict math to handle this problem. `Math` has been extended by a couple of methods who all ends with `exact`, e.g. `addExact`. Those methods handle overflows properly by throwing an `ArithmeticException` when the result of the operation doesn't fit into the number type:

```
try {
    Math.addExact(Integer.MAX_VALUE, 1);
}
catch (ArithmeticException e) {
    System.err.println(e.getMessage());
    // => integer overflow
}
```

The same exception might be thrown when trying to convert longs to int via `toIntExact`:

```
try {
    Math.toIntExact(Long.MAX_VALUE);
}
catch (ArithmeticException e) {
    System.err.println(e.getMessage());
    // => integer overflow
}
```

Working with Files

The utility class `Files` was first introduced in Java 7 as part of Java NIO. The JDK 8 API adds a couple of additional methods which enables us to use functional streams with files. Let's deep-dive into a couple of code samples.

Listing files

The method `Files.list` streams all paths for a given directory, so we can use stream operations like `filter` and `sorted` upon the contents of the file system.

```
try (Stream<Path> stream = Files.list(Paths.get(""))) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> !path.startsWith("."))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("List: " + joined);
}
```

The above example lists all files for the current working directory, then maps each path to it's string representation. The result is then filtered, sorted and finally joined into a string. If you're not yet familiar with functional streams you should read my Java 8 Stream Tutorial.

You might have noticed that the creation of the stream is wrapped into a try/with statement. Streams implement `AutoCloseable` and in this case we really have to close the stream explicitly since it's backed by IO operations.

> The returned stream encapsulates a DirectoryStream. If timely disposal of file system resources is required, the try-with-resources construct should be used to ensure that the stream's close method is invoked after the stream operations are completed.

Finding files

The next example demonstrates how to find files in a directory or it's sub-directories.

```
Path start = Paths.get("");
int maxDepth = 5;
try (Stream<Path> stream = Files.find(start, maxDepth, (path, attr) ->
        String.valueOf(path).endsWith(".js"))) {
    String joined = stream
        .sorted()
        .map(String::valueOf)
        .collect(Collectors.joining("; "));
    System.out.println("Found: " + joined);
}
```

The method `find` accepts three arguments: The directory path `start` is the initial starting point and `maxDepth` defines the maximum folder depth to be searched. The third argument is a matching predicate and defines the search logic. In the above example we search for all JavaScript files (filename ends with .js).

We can achieve the same behavior by utilizing the method `Files.walk`. Instead of passing a search predicate this method just walks over any file.

```
Path start = Paths.get("");
int maxDepth = 5;
try (Stream<Path> stream = Files.walk(start, maxDepth)) {
    String joined = stream
        .map(String::valueOf)
        .filter(path -> path.endsWith(".js"))
        .sorted()
        .collect(Collectors.joining("; "));
    System.out.println("walk(): " + joined);
}
```

In this example we use the stream operation `filter` to achieve the same behavior as in the previous example.

Reading and writing files

Reading text files into memory and writing strings into a text file in Java 8 is finally a simple task. No messing around with readers and writers. The method `Files.readAllLines` reads all lines of a given file into a list of strings. You can simply modify this list and write the lines into another file via `Files.write` :

```
List<String> lines = Files.readAllLines(Paths.get("res/nashorn1.js"));
lines.add("print('foobar');");
Files.write(Paths.get("res/nashorn1-modified.js"), lines);
```

Please keep in mind that those methods are not very memory-efficient because the whole file will be read into memory. The larger the file the more heap-size will be used.

As an memory-efficient alternative you could use the method `Files.lines` . Instead of reading all lines into memory at once, this method reads and streams each line one by one via functional streams.

```
try (Stream<String> stream = Files.lines(Paths.get("res/nashorn1.js"))) {
    stream
        .filter(line -> line.contains("print"))
        .map(String::trim)
        .forEach(System.out::println);
}
```

If you need more fine-grained control you can instead construct a new buffered reader:

```
Path path = Paths.get("res/nashorn1.js");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    System.out.println(reader.readLine());
}
```

Or in case you want to write to a file simply construct a buffered writer instead:

```
Path path = Paths.get("res/output.js");
try (BufferedWriter writer = Files.newBufferedWriter(path)) {
    writer.write("print('Hello World');");
}
```

Buffered readers also have access to functional streams. The method `lines` construct a functional stream upon all lines denoted by the buffered reader:

```
Path path = Paths.get("res/nashorn1.js");
try (BufferedReader reader = Files.newBufferedReader(path)) {
    long countPrints = reader
        .lines()
        .filter(line -> line.contains("print"))
        .count();
    System.out.println(countPrints);
}
```

So as you can see Java 8 provides three simple ways to read the lines of a text file, making text file handling quite convenient.

Unfortunately you have to close functional file streams explicitly with try/with statements which makes the code samples still kinda cluttered. I would have expected that functional streams auto-close when calling a terminal operation like `count` or `collect` since you cannot call terminal operations twice on the same stream anyway.

I hope you've enjoyed this article. All code samples are hosted on GitHub along with plenty of other code snippets from all the Java 8 articles of my blog. If this post was kinda useful to you feel free to star the repo and follow me on Twitter.

Keep on coding!

Avoiding Null Checks in Java 8

March 15, 2015

How to prevent the famous `NullPointerException` in Java? This is one of the key questions every Java beginner will ask sooner or later. But also intermediate and expert programmers get around this error every now and then. It's by far the most prevalent kind of error in Java and many other programming languages as well.

Tony Hoare, the inventor of the null reference apologized in 2009 and denotes this kind of errors as his **billion-dollar mistake**.

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Anyways, we have to deal with it. So what can we do to prevent NullPointerExceptions at all? Well, the obvious answer is to add null checks all around the place. Since null checks are kinda cumbersome and painful many languages add special syntax for handling null checks via null coalescing operators - also known as *elvis operator* in languages like Groovy or Kotlin.

Unfortunately Java doesn't provide such a syntactic sugar. But luckily things get better in Java Version 8. This post describes a couple of techniques how to prevent writing needless null checks by utilizing new features of Java 8 like lambda expressions.

Improving Null Safety in Java 8

I've already shown in another post how we can utilize the `Optional` type of Java 8 to prevent null checks. Here's the example code from the original post.

Assuming we have a hierarchical class structure like this:

```
class Outer {
    Nested nested;
    Nested getNested() {
        return nested;
    }
}
class Nested {
    Inner inner;
    Inner getInner() {
        return inner;
    }
}
class Inner {
    String foo;
    String getFoo() {
        return foo;
    }
}
```

Resolving a deep nested path in this structure can be kinda awkward. We have to write a bunch of null checks to make sure not to raise a `NullPointerException`:

```
Outer outer = new Outer();
if (outer != null && outer.nested != null && outer.nested.inner != null) {
    System.out.println(outer.nested.inner.foo);
}
```

We can get rid of all those null checks by utilizing the Java 8 `Optional` type. The method `map` accepts a lambda expression of type `Function` and automatically wraps each function result into an `Optional`. That enables us to pipe multiple `map` operations in a row. Null checks are automatically handled under the hood.

```
Optional.of(new Outer())
    .map(Outer::getNested)
    .map(Nested::getInner)
    .map(Inner::getFoo)
    .ifPresent(System.out::println);
```

An alternative way to achieve the same behavior is by utilizing a supplier function to resolve the nested path:

```
Outer obj = new Outer();
resolve(() -> obj.getNested().getInner().getFoo());
    .ifPresent(System.out::println);
```

Calling `obj.getNested().getInner().getFoo())` might throw a `NullPointerException`. In this case the exception will be caught and the method returns `Optional.empty()`.

```
public static <T> Optional<T> resolve(Supplier<T> resolver) {
    try {
        T result = resolver.get();
        return Optional.ofNullable(result);
    }
    catch (NullPointerException e) {
        return Optional.empty();
    }
}
```

Please keep in mind that both solutions are probably not as performant as traditional null checks. In most cases that shouldn't be much of an issue.

As usual the above code samples are hosted on GitHub.

Happy coding!

> UPDATE: I've updated the code samples thanks to a hint from Zukhramm on Reddit.

Fixing Java 8 Stream Gotchas with IntelliJ IDEA

March 05, 2015

Java 8 has been released almost one year ago in March 2014. At Pondus we've managed to update all of our production servers to this new version back in May 2014. Since then we've migrated major parts of our code base to lambda expressions, streams and the new Date API. We also use Nashorn to dynamically script parts of our application which may change during runtime.

The most used feature besides lambdas is the new Stream API. Collection operations are all around the place in almost any codebase I've ever seen. And Streams are a great way to improve code readability of all those collection crunching.

But one thing about streams really bothers me: Streams only provide a few terminal operations like `reduce` and `findFirst` directly while others are only accessible via `collect`. There's a utility class Collectors, providing a bunch of convenient collectors like `toList`, `toSet`, `joining` and `groupingBy`.

For example this code filters over a collection of strings and creates a new list:

```
stringCollection
    .stream()
    .filter(e -> e.startsWith("a"))
    .collect(Collectors.toList());
```

After migrating a project with 300k lines of code to streams I can say that `toList`, `toSet` and `groupingBy` are by far the most used terminal operations in our project. So I really cannot understand the design decision not to integrate those methods directly into the `Stream` interface so you could just write:

```
stringCollection
    .stream()
    .filter(e -> e.startsWith("a"))
    .toList();
```

This might look like a minor imperfection at first but it gets really annoying if you have to use this kind of stuff over and over again.

There's a method `toArray()` but no `toList()`. So I really hope some of the more convenient collectors will make it's way into the `Stream` interface in Java 9. Brian? □_□

> As a side note: Stream.js is a JavaScript port of the Java 8 Streams API for the browser and addresses the described issue nicely. All important terminal operations are directly accessible on the stream itself for convenience. See the API doc for details.

Anyways. IntelliJ IDEA claims to be the most intelligent Java IDE. So let's see how we can utilize IDEA to solve this problem for us.

IntelliJ IDEA to the rescue

IntelliJ IDEA comes with a handy feature called Live Templates. If you don't already know what it is: Live Templates are shortcuts for commonly used code snippets. E.g. you type `sout` + tabulator and IDEA inserts the code snippet `System.out.println()`. Read here to learn more about it.

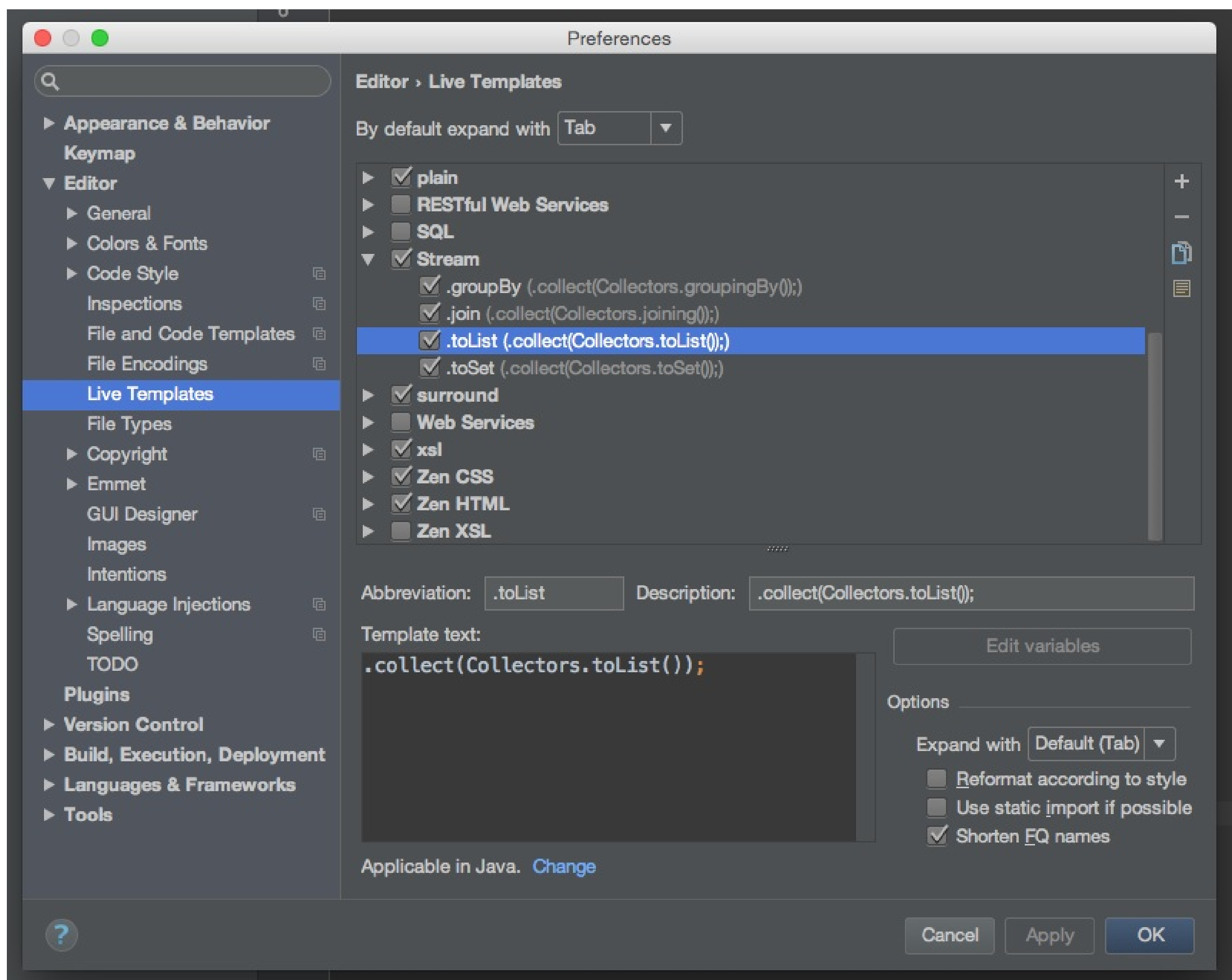
How does Live Templates help with the problem described above? Actually we can simply create our own Live Templates for all the commonly used default Stream collectors. E.g. we can create a Live Template with the abbreviation `.toList` to insert the appropriate collector `.collect(Collectors.toList())` automatically.

This is how it looks like in action:



Set up your own Live Templates

Let's see how we can set this up. First go to Settings and choose Live Templates in the menu to the left. You can also use the handy filter input at the top left of the dialog.



Next we can create a new group called `Stream` via the `+` icon on the right. Next we add all of our stream-related Live Templates to this group. I'm using the default collectors `toList`, `toSet`, `groupingBy` and `join` quite commonly, so I create a new Live Template for each of those methods.

This part is **important**: After adding a new Live Template you have to specify the applicable context at the bottom of the dialog. You have to choose *Java* → *Other*. Afterwards you define the abbreviation, a description and the actual template code.

```
// Abbreviation: .toList
.collect(Collectors.toList())

// Abbreviation: .toSet
.collect(Collectors.toSet())

// Abbreviation: .join
.collect(Collectors.joining("$END$"))

// Abbreviation: .groupBy
.collect(Collectors.groupingBy(e -> $END$))
```

The special variable `$END$` determines the cursors position after using the template, so you can directly start typing at this position, e.g. to define the joining delimiter.

> Hint: You should enable the option "Add unambiguous imports on the fly" so IDEA automatically adds an import statement to `java.util.stream.Collectors`. The option is located in: *Editor → General → Auto Import*

Let's see those two templates in action:

Join

```
stringCollection
        .stream()
        .filter(s -> s.startsWith("a"))
        |
```

GroupBy

```
stringCollection
        .stream()
        .filter(s -> s.startsWith("a"))
```

Live Templates in IntelliJ IDEA are an extremely versatile and powerful tool. You can greatly increase your coding productivity with it. Do you know other examples where Live Templates can save your live? Let me know!

Still not satisfied? Learn everything you ever wanted to know about Java 8 Streams in my Streams Tutorial.

Happy coding.

Using Backbone.js with Nashorn

April 07, 2014

This example demonstrates how to use Backbone.js models with the Java 8 Nashorn Javascript Engine. First released in March 2014 as part of Java SE 8, Nashorn extends Javas capabilities by running javascript code natively on the JVM. For java web developers Nashorn might be especially useful for re-using existing client-side code on the java server. Traditionally Node.js was in a clear advantage, but Nashorns possibilities might close the gap to the JVM.

When working with modern javascript MVC frameworks like Backbone.js for HTML5 front-ends, more and more code moves from the server back-end to the web front-end. This approach can greatly increase the user experience because you safe a lot of server-roundtrips when using business logic from your views.

Backbone enables you to define model classes which can be bound to views (e.g. HTML forms). Backbone keeps track of updating the model when the user interacts with the UI and vice versa. It also aids you by synchronizing your model with the server, e.g. by calling the appropriate method of your REST handler on the server side. So you end up implementing business logic in your front-end code, leaving your server model responsible for persisting data.

Reusing backbone models on the server side is quite easy with Nashorn, as the following example will demonstrate. Before we start make sure you're familiar with writing javascript for the Nashorn Engine by reading my Nashorn Tutorial.

The Java Model

First, we define a domain class `Product` in java code. This class might be used for CRUD database operations (saving to and loading from a datasource). Keep in mind that this class is a dumb Java Bean without any business logic applied, because we want our front-end to be capable of executing the business logic right from the UI.

```
class Product {
    String name;
    double price;
    int stock;
    double valueOfGoods;
}
```

The Backbone Model

Now we define the backbone model as the counter-part of our java bean. The backbone model `Product` uses the same data-structure as the java bean, since this is the data we might want to persist on the java server.

The backbone model also implements the business logic: The method `getValueOfGoods` calculates the value of all products by multiplying `stock` with `price`. Each time `stock` or `price` changes the property `valueOfGoods` must be re-calculated.

```
var Product = Backbone.Model.extend({
    defaults: {
        name: '',
        stock: 0,
        price: 0.0,
        valueOfGoods: 0.0
    },

    initialize: function() {
        this.on('change:stock change:price', function() {
            var stock = this.get('stock');
            var price = this.get('price');
            var valueOfGoods = this.getValueOfGoods(stock, price);
            this.set('valueOfGoods', valueOfGoods);
        });
    },

    getValueOfGoods: function(stock, price) {
        return stock * price;
    }
});
```

Since the backbone model doesn't use any Nashorn language extensions, we can safely use the same code both on the client (Browser) and on the server (Java) side.

Keep in mind that I deliberately chose a really simple function for demonstrating purposes only. Real business logic should be more complex.

Putting both together

The next goal is to re-use the backbone model from Nashorn, e.g. on the java server. We want to achieve the following behavior: bind all properties from the java bean to the backbone model; calculate the property `valueOfGoods`; pass the result back to java.

First, we create a new script to be evaluated solely by Nashorn, so we can safely use Nashorn extensions here:

```
load('http://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.6.0/underscore-min.js');
load('http://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.1.2/backbone-min.js');
load('product-backbone-model.js');

var calculate = function(javaProduct) {
    var model = new Product();
    model.set('name', javaProduct.name);
    model.set('price', javaProduct.price);
    model.set('stock', javaProduct.stock);
    return model.attributes;
};
```

The script first loads the relevant external scripts Underscore and Backbone (Underscore is a pre-requirement for Backbone) and our `Product` backbone model as defined above.

The function `calculate` accepts a `Product` java bean, binds all properties to a newly created backbone `Product` and returns all attributes of the model back to the caller. By setting the properties `stock` and `price` on the backbone model, property `valueOfGoods` will automatically be calculated due to the event handler registered in the models `initialize` constructor function.

Finally, we call the function `calculate` from java:

```
Product product = new Product();
product.setName("Rubber");
product.setPrice(1.99);
product.setStock(1337);

ScriptObjectMirror result = (ScriptObjectMirror)
    invocable.invokeFunction("calculate", product);

System.out.println(result.get("name") + ": " + result.get("valueOfGoods"));
// Rubber: 2660.63
```

We create a new `Product` java bean and pass it to the javascript function. As a result the method `getValueOfGoods` will be triggered, so we can read the property `valueOfGoods` from the returning object.

Conclusion

Reusing existing javascript libraries on the Nashorn Engine is quite easy. Backbone is great for building complex HTML5 front-ends. In my opinion Nashorn and the JVM now is a great alternative to Node.js, since you can make use of the whole Java eco-system in your Nashorn codebase, such as the whole JDK API and all available libraries and tools. Keep in mind that you're not tight to the Java Language when working with Nashorn - think Scala, Groovy, Clojure or even pure Javascript via `jjs`.

The runnable source code from this article is hosted on GitHub (see this file). Feel free to fork the repository or send me your feedback via Twitter.